

DTIC File Copy

2

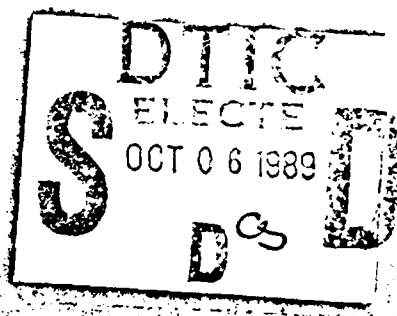
Ada 9X Project Report



AD-A213 503

Ada 9X Project Revision Request Report

August 1989



Office of the Under Secretary of Defense for Acquisition

Washington, D. C. 20301

Approved for public release; distribution is unlimited.

89 10 6 0 23

Ada 9X Project Report



Ada 9X Project Revision Request Report

August 1989

Accession For	
NTIS - OMA	J
DTIC - OMA	
Uncl. - OMA	
Just. - OMA	
By	
Dist	
A-1	

Office of the Under Secretary of Defense for Acquisition

Washington, D. C. 20301

Approved for public release; distribution is unlimited.

PREFACE

This document contains the Ada 9X revision requests submitted to the Ada Joint Program Office (AJPO) as of 25 July 1989. These requests are being considered for inclusion in the current revision of ANSI/MIL-STD-1815A by the Ada 9X Project Requirements Team. Updates of this document will be issued periodically. The revision requests can also be viewed on the Ada 9X electronic bulletin board at 1-800-Ada9X25 or 1-301-459-8939 and in Europe on Eurokom.

✓ This document was prepared by IIT Research Institute (IITRI) under sponsorship of the AJPO. The IITRI Program Manager is Ms. Mary Armstrong (IITRI, 4600 Forbes Boulevard, Lanham, Maryland 20706). Other IITRI staff that contributed to the document are Mr. Hank Greene, Ms. Susan Carlson, and Ms. Trisha Guethlein. The AJPO Program Manager is Ms. Christine Anderson, the Ada 9X Project Manager (Air Force Armament Laboratory/FXG, Eglin Air Force Base, Florida 32542-5434).

INTRODUCTION

BACKGROUND

The overall goal of the Ada 9X Project is to revise ANSI/MIL-STD-1815A to reflect current essential requirements with minimum negative impact and maximum positive impact to the Ada community. The Ada 9X process is a revision and not a redesign of the language and should be viewed as a natural part of the maturation process.

Requirements for the revision are based on input from the Ada community in the form of special studies, workshops, public meetings, Ada Language Issues, and revision requests. The solicitation of revision requests was initiated in October 1988 and will continue through October 1989. Individuals and groups are encouraged to submit requests for a particular revision of the language using the format shown in Appendix A. Revision requests are being reviewed by the Ada 9X Project Requirements Team. The status of revision requests will be tracked throughout the Ada 9X revision process. Revision requests may be viewed on the Ada 9X Project electronic bulletin board at 1-800-Ada9X25 or 1-301-459-8939 and in Europe on Eurokom.

ORGANIZATION OF THIS DOCUMENT

This document contains all the revision requests submitted as of 25 July 1989. Requests are organized according to relevant sections in the Language Reference Manual, ANSI/MIL-STD-1815A. If a request is relevant to more than one section, it will physically appear in the first section designated by the author as a reference, and appear by reference in other sections. Indices are provided to support referencing by key words, submitter, organization, and revision request number. This document will be updated and reissued periodically throughout the duration of the Ada 9X Project.

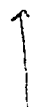


TABLE OF CONTENTS

1.	INTRODUCTION	1-1
2.	LEXICAL ELEMENTS	2-1
3.	DECLARATION AND TYPES	3-1
4.	NAMES AND EXPRESSIONS	4-1
5.	STATEMENTS	5-1
6.	SUBPROGRAMS	6-1
7.	PACKAGES	7-1
8.	VISIBILITY RULES	8-1
9.	TASKS	9-1
10.	PROGRAM STRUCTURE AND COMPILATION ISSUES	10-1
11.	EXCEPTIONS	11-1
12.	GENERIC UNITS	12-1
13.	REPRESENTATION CLAUSES AND IMPLEMENTATION-DEPENDENT FEATURES	13-1
14.	INPUT-OUTPUT	14-1
15.	REVISION REQUEST THAT REFERENCE AN ANSI/MIL-STD-1815A ANNEX OR APPENDIX	15-1
16.	REVISION REQUESTS THAT DO NOT REFERENCE ANSI/MIL-STD-1815A	16-1

APPENDIX

A.	SAMPLE REVISION REQUEST FORM	A-1
----	------------------------------	-----

INDICES

KEY TECHNICAL TERMS	I-2
REVISION REQUEST BY NUMBER	I-11
REVISION REQUEST BY TITLE	I-18
REVISION REQUEST BY SUBMITTER	I-25
REVISION REQUEST BY ORGANIZATION	I-28

**ADA 9X REVISION REQUESTS
THAT REFERENCE
ANSI/MIL-STD-1815A**

SECTION 1. INTRODUCTION

STANDARDIZATION OF GENERAL PURPOSE PACKAGES**DATE:** February 10, 1989**NAME:** Mark F. Komatar**ADDRESS:** U.S. Army Management Engineering College
ATTN: AMXOM-PA (M. Komatar)
Rock Island, IL 61299-7040**TELEPHONE:** (301) 782-4041
AUTOVON: 793-4041**ANSI/MIL-STD-1815A REFERENCE:** 1.1.1(5)**PROBLEM:**

All mature, production-quality computer languages include standard (or more-or-less standard) libraries of useful, general-purpose routines -- except Ada. Examples of such routines include: square-root, sin, cos, and other math functions; string edit functions, money-type operators, and access methods interfaces. As a result of this lack, Ada programmers are continually re-inventing many software wheels, or are digging through Ada (non-standard) software repositories looking for general-purpose routines, or are tying their code to non-standard packages supplied by their compiler vendors. The problem: Inefficient implementations of general-purpose routines (math packages, string packages, etc.) and/or non-portable code.

IMPORTANCE: ESSENTIAL

One of the basic tenets of the Ada philosophy is strict standardization. If common math, string-handling, and other useful routines are not standardized (in the form of a package specifications at the language-definition level), then potential billions of dollars will be wasted writing inefficient and/or non-portable Ada code.

CURRENT WORKAROUNDS:

There are no workarounds; all current "workarounds" are part of the problem: application programmers writing inefficient implementations of routines that they simply take for granted in other mature languages; multiple Ada software repositories where general-purpose packages are mixed with literally thousands of highly-specialized ones -- and the problems inherent in searching, accessing, retrieving and testing software from such repositories; and finally, compiler vendors writing efficient but non-portable (across compilers) implementations of general-purpose routines.

POSSIBLE SOLUTIONS:

The solution is to have DoD, ANSI, and ISO research and define package specifications of those common, general-purpose routines most needed to make Ada a truly mature, application-oriented language. The specific change to ANSI/MIL-STD-1815A might be an additional chapter similar in format to Chapter 14, "Input-Output". Once such specifications are standardized, then Ada compiler vendors can compete to provide the most efficient implementations of said packages.

INCORRECT ORDER DEPENDENCIES**DATE:** February 20, 1989**NAME:** Mats Weber**ADDRESS:** Swiss Federal Institute of Technology
EPFL DI LITH
1015 Lausanne
Switzerland**TELEPHONE:** 0041 21 693 42 14
E-mail: madmats@elma.epfl.ch**ANSI/MIL-STD-1815A REFERENCE:** 1.6(9)**PROBLEM:**

LRM 1.6(9) says : "... Furthermore, the construct is incorrect if execution of these parts in a different order would have a different effect."

First, we all know that no compiler will ever be able to detect all such errors (because it leads at least to NP-hard, if not undecidable problems).

Second, the LRM does not explain exactly what "would have a different effect" means. For example, consider the following example on an implementation where System.Address is an integer type:

```
with System,
  Unchecked_Conversion,
  Text_IO;

procedure Is_This_An_Incorrect_Order_Dependence is

  type Access_Integer is access Integer;

  function To_Address is new Unchecked_Conversion(Access_Integer,
    System.Address);

  type Rec is
    record
      A : Access_Integer;
      B : Access_Integer;
    end record;

  X : constant Rec := (A => new Integer, B => new Integer);
  -- The order in which the allocators are evaluated is
  -- not defined by the language.

begin
```



```
if System."<"(To_Address(X.A), To_Address(X.B)) then
  Text_IO.Put_Line("A < B");
else
  Text_IO.Put_Line("B < A");
end if;
end Is_This_An_Incorrect_Order_Dependence;
```

OK, this is a bad example because it uses `Unchecked_Conversion` and `System.Address`, but it is still legal Ada. The question is: does the above example contain an incorrect order dependence? If the answer is yes, it means that two or more allocators should never appear in an aggregate.

Now suppose we change `Text_IO.Put_Line("B < A");` (in the else part of the if statement) to `Text_IO.Put_Line("A < B");` The output of the program would be "A < B" in any case, but two paths through the program are possible. Is this considered to be "a different effect"?

IMPORTANCE: IMPORTANT

I think that it should be possible for an Ada compiler to decide if an Ada program is correct or not (at compilation time or at runtime). Uninitialized variables is a detectable problem (at runtime) but incorrect order dependence is not.

Many programmers are unaware of the existence of incorrect order dependence and erroneous construct. They think that "If my program compiles and executes fine, then it is a legal program".

CURRENT WORKAROUNDS:

POSSIBLE SOLUTIONS:

- Define what "having a different effect" means in the 9X LRM (this is necessary, but would not solve the problem).
- Allow programs with incorrect order dependence to execute in different ways without being illegal. But don't allow raising `Program_Error`.
- Explicitly state the order in which different parts of a construct are executed (order of evaluation of actual parameters, components of aggregates, etc..., but NOT the execution of tasks).

This is a bad solution because changing the order of declaration of components of a record or formal parameters of a subprogram could change the effect of a program. But it has the advantage that the effect of a program is well defined.

ERRONEOUS EXECUTION AND INCORRECT ORDER DEPENDENCE**DATE:** March 21, 1989**NAME:** B. A. Wichmann (from material supplied by members of Ada UK)**ADDRESS:** National Physical Laboratory
Teddington, Middx
TW11 OLW
UK**TELEPHONE:** +44 1 943 6976, (Messages: +44 1 977 3222,
E-mail: baw@seg.npl.co.uk)**ANSI/MIL-STD-1815A REFERENCE:** 1.6(c), 1.6(d)**PROBLEM:**

There can be no doubt that the existence of both erroneous programs and ones having incorrect order dependence is highly undesirable. Programs with either property will execute differently on conforming, validated systems. The original language design acknowledged that neither property could be totally avoided in a language having the expressive power of Ada.

Given that both properties will continue to exist in any revised language does not mean that the risks associated with these cannot be reduced. The time has come to make an in-depth study of the issues and to propose mechanisms to reduce the risks for practical Ada programs.

The problems with both the above properties arise because implementations of Ada need to vary to suit the underlying architecture. Such differences are well accepted for all programming languages - the only special feature of Ada in this respect is that the comprehensive nature of the language means that there are several complex issues to resolve.

The Ada Rapporteur Group has identified some ambiguities in the situations which lead to an erroneous execution and has resolved these within the current language design.

However, the work that has just started of the Uniformity Rapporteur Group (URG) is more relevant to this issue. Guidance to implementors on the preferred strategies for handling erroneous programs does reduce the risks to users. But relatively small changes to the language definition could dramatically reduce the risks to manageable proportions.

As an example of the nature of the problems, consider the handling of an undefined simple variable. This is technically erroneous which implies that no semantics is given to the program. If detected by an implementation, it could raise `PROGRAM_ERROR`, while without detection anything could happen. Proposals by the URG would restrict the actions by the implementation so that programs could recover by means of the exception mechanism.

A further example is given by the freedom of implementation to choose the mechanism for parameter passing. While passing large objects by reference is almost universal now, to require this would be a mistake since passing by copy is required on multiprocessors without shared memory. If a program depends upon the parameter mechanism, it is erroneous even if it would function correctly (but differently) on copy

on reference implementations. It would seem that such programs could be allowed to have different effects while not being erroneous. Hence an implementation could still use either mechanism, but the semantics would be either copy or reference (for any specific call). The freedom given by the current language is far more than implementors require. For instance, as it stands, an implementation could choose the mechanism dynamically, even if the subprogram was compiled in-line.

IMPORTANCE: **IMPORTANT**

But **ESSENTIAL** for safety-critical software.

CURRENT WORKAROUNDS:

In the long-term, program analysis tools could make some impact on the problem. No effort is known by the author to detect erroneous programs for the full language. Hence there are not effective workarounds known.

POSSIBLE SOLUTIONS:

It is felt that a significant improvement over the current position could be attained by requiring implementations to define critical properties related to this area. For instance, the implementation could be required to define the parameter mechanism used, or define the order of elaboration of library units.

ISO WG9 be asked to study the implications of erroneous execution and incorrect order dependence upon the effective use of Ada.

ISO WG9 be asked to include in its agreed work item on the uniformity of implementations to prepare proposals for Ada 9X to reduce the impact of erroneous programs and those with incorrect order dependence.

**ADA 9X REVISION REQUESTS
THAT REFERENCE
ANSI/MIL-STD-1815A**

SECTION 2. LEXICAL ELEMENTS

USE 8-BIT ASCII

DATE: January 14, 1989

NAME: William Thomas Wolfe

ADDRESS: Home: 102 Edgewood Avenue #2
Clemson, SC 29631

Office: Department of Computer Science
Clemson University
Clemson, SC 29634

TELEPHONE: Home: (803) 654-7030
Office: (803) 656-2847
E-mail: wtwolfe@hubcap.clemson.edu

ANSI/MIL-STD-1815A REFERENCE: 2.1(1)

PROBLEM:

Ada is based on 7-bit ASCII (ISO 646), rather than on 8-bit ASCII (ISO 8859/1-9).

CONSEQUENCES:

It is difficult to construct programs which are capable of handling the characters which appear in European languages other than English; this leads to English-language dependencies or to machine dependencies and non-portability in Ada-based products.

IMPORTANCE:**CURRENT WORKAROUNDS:**

Ada-based products can be made machine-dependent and non-portable.

POSSIBLE SOLUTIONS:

Revise ANSI/MIL-STD-1815A 2.1 (1) to read:

"The only characters allowed in the text of a program are the graphic characters and the format effectors. Each graphic character corresponds to a unique code of the ISO eight-bit coded character set (ISO standards 8859/1-9), and is represented visually by a graphic symbol. The description of the language definition in this standard reference manual uses the ASCII graphic symbols, the ANSI graphical representation of the ISO character set."

UNDERSCORE BEFORE EXPONENT IN NUMERIC LITERALS**DATE:** May 23, 1989**NAME:** Jurgen F H Winkler**ADDRESS:** Siemens AG ZFE F2 SOF3
Otto-Hahn-Ring 6
D-8000 Munchen 83
Fed Rep of Germany**TELEPHONE:** +49 89 636 2173**ANSI/MIL-STD-1815A REFERENCE:** 2.4**PROBLEM:**

Allow underscore before exponent part of numeric literals

Ada allows the underscore in numeric literals. This can be used to improve readability:

1000000 vs. 1_000_000.

Readability of literals containing an exponent part is impaired by the fact that the underscore is not allowed before the exponent part:

1.34E-12 vs. 1.34_E-12

IMPORTANCE: ADMINISTRATIVE**CURRENT WORKAROUNDS:****POSSIBLE SOLUTIONS:**exponent::= [integer] E [+] integer | [integer] E - integer

INTERFACING FORTRAN LIBRARIES TO ADA PROGRAMS

DATE: January 2, 1989

NAME: C. G. Van der Laan

ADDRESS: Rekencentrum der Rijksuniversiteit
Landleven 1
Postbus 800
9700 Av Groningen
The Netherlands

TELEPHONE: -50-633374
E-mail: cgl@HGRRUGS (bilnet)

ANSI/MIL-STD-1815A REFERENCE: 2.8

PROBLEM:

Interfacing FORTRAN libraries to Ada programs.

IMPORTANCE: IMPORTANT

If not satisfied, FORTRAN numerical libraries can be used marginally.

CURRENT WORKAROUNDS:

Interface pragma, but its use is limited (see referenced report)

POSSIBLE SOLUTIONS:

Van der Laan. 1985. "Interfacing Ada to FORTRAN", Rekencentrum der rijksuniversiteit te groningen.

1. INTRODUCTION

Use of FORTRAN routines in an Ada environment is an example of mixed language programming. Mixed language programming came into existence along with the introduction of high-level algorithmic programming languages via the use of assembly routines in high-level languages. Nowadays, mixed language programming with modern high-level languages and FORTRAN is interesting because of the general availability of FORTRAN numerical program libraries on various machine ranges (e.g., NAG mark 10 on roughly 52 computer series and IMSL edition 9 on roughly 25 computer series, while PORT-conforming routines, such as CALGO, are transportable and can be used on practically any computer). Mixed languages programming activities are described in Einarsson & Gentleman (1984), Van der Laan (1981, 1983a), De Bruin & Van der Lann (1983), while pioneering work is described by Gentleman and Traub (1968). A coherent approach is given by the UNIX environment, where programs written in C, FORTRAN, and PASCAL can be intermixed, with C an intermediate language (Feldman (1981)); another example of this direction is given by the UCSD-P system. The problems with the latter possibilities is that they are hardly documented. Related activities are concerned with preprocessors (Feldman (1983)) and language transformation (Partsch & Steinbrugger (1981)). The advantages of using FORTRAN program libraries in other languages are:

- extensive, efficient, fully documented, thoroughly (field) tested FORTRAN libraries exist for many

computers.

- maintenance of program libraries with transliterations in various languages is cumbersome.
- The disadvantages are:

- use of FORTRAN routines in other languages is generally not elaborated and therefore not standardized and hardly provided by manufacturers,
- dealing with more than one language is more complex and error prone,
- portability is decreased by mixed language programming, unless (trans-) portable, or generally available, operating systems or programming environments are introduced as environments for compatible compilers,
- error-handling, and input-output in general, is more complicated;
- interfacing overhead, especially for small routines;
- fewer protection mechanisms are available in FORTRAN because it is not a strongly-typed language.

Difficulties with mixed language programming are discussed by Einarsson & Gentleman (1984), who distinguish fundamental incompatibilities between languages from incompatible implementations of languages. In the Ada Reference Manual (ARM in the sequel) the INTERFACE pragma for interfaces with other languages, is dealt with in section 13.9; the general pragma syntax is given in section 2.8. Elaborated pragmas with respect to interfaces with other languages can be supplied in appendix F of the ARM. In this paper we explore the INTERFACE pragma with respect to the creation of virtual numerical program collections in Ada on top of FORTRAN program libraries. Moreover, we specify a FORTRAN pragma, in order to couple routines which have subprograms as dummy arguments. Our proposal is within the syntax of the standard because we believe that modification of the standard will at least take time, if it happens at all. We hope this paper will inspire manufacturers or software houses to implement the INTERFACE pragma as well as the tailored FORTRAN pragma, in order to make the use of FORTRAN program libraries in Ada possible; furthermore we hope that the specification of the FORTRAN pragma will be included in appendix F of ARM. This work is a continuation of our earlier experiences in mixed language programming with ALGO 68 and FORTRAN (Van der Lann (1981), De Bruin and Van der Laan (1983)); so far we are only able to describe our ideas because we have no access to an Ada compiler. Moreover, the FORTRAN pragma is a proposal which is not yet implemented to our knowledge and not mentioned in ARM. However, our examples are represented as compilation units, especially packages, so they can be tried out when a sufficiently sophisticated compiler can be accessed.

Notational conventions.

ARM denotes the Ada reference manual; FRM denotes the FORTRAN reference manual. In the examples the following naming conventions are used. If we start from a FORTRAN program library collection with name X then the FORTRAN bound interface level is called XF, the Ada bound interface level is supplied as a package specification with name X_A, while the virtual Ada collection is supplied as a package with name VIR_X. The latter layer is not specific for the interface but supplied in order to elucidate the X_A level. Packages were chosen as units because we like to couple collections, with encapsulated data types, instead of single routines; furthermore, the designators of routines as library units are restricted to identifiers. GEF stands for the collection of subprograms published by Forsythe et al. (1977).

4. THE FORTRAN PRAGMA

In order to make use in Ada of FORTRAN routines with Ada subprograms as parameters, we propose a FORTRAN pragma. This pragma obeys the general syntax of pragmas as given in section 2.8 of the ARM, and quoted below:

```
pragma ::= pragma identifier [( argument_association
                               {, argument_association} )];
```



```
argument_association ::= [ argument_identifier =>] name |
                        [ argument_identifier =>] expression. "
```

We propose the FORTRAN pragma

```
pragma FORTRAN (NAMEF [( A1 [( B1, aggregate { , BM, aggregate} )
                        { , AR [( BR1, aggregate { , BRM, aggregate} )
                        ]
                        ]
                        )
                        ]
                )
```

where A1, ..., AR correspond to the dummy arguments of NAMEF, and B1, ..., BM, and BR1,...BRM correspond to arrays of a dummy subprogram, with bounds given in the subsequent aggregate (for each dimension lower and upper bounds must be given, all separated by commas), of the formal generic subprograms. NAMEF denotes the name of the coupled FORTRAN subprogram.

Name restrictions are imposed by those of FORTRAN. The FORTRAN pragma is an extension of the INTERFACE pragma and aimed at generic subprograms with only procedures or functions as generic formal parameters; as extension of the INTERFACE pragma it may be used for the cases treated in section 3, by substitution of

```
pragma INTERFACE( FORTRAN, NAME) by pragma FORTRAN( NAME),
```

It is a pity that in Ada subprograms are allowed as parameters of subprograms only via generic units. A large class of numerical problems such as : quadrature, zero finding, optimization, and solving differential and integral equations, parameterize over a function. Routines in FORTRAN for the above problems parameterize in general via subprograms as dummy arguments. When a dummy subprogram itself contains an array as dummy argument, one must supply information about the bounds, because FORTRAN generally lacks this information. (This is done via the above mentioned aggregates.)

It must be kept in mind that use of the FORTRAN pragma with generic units severely restricts the generic formal parameters: together with the FORTRAN pragma only functions and subprograms are allowed as generic formal parameters, because that is what we need in order to couple FORTRAN subprograms with dummy subprograms and it will probably relieve the implementation of the FORTRAN pragma.

For additional references to Section 2. of ANSI/MIL-STD-1815A, see the following sections, revision request numbers, and revision request titles in this document.

REVISION REQUEST

<u>NUMBER</u>	<u>TITLE</u>	<u>SECTION</u>
0049	REFERENCE TO VARIABLE NAMES	5

**ADA 9X REVISION REQUESTS
THAT REFERENCE
ANSI/MIL-STD-1815A**

SECTION 3. DECLARATION AND TYPES

PRE-ELABORATION

DATE: June 7, 1989

NAME: Ted Baker (ACM Special Interest Group on Ada, Ada Runtime Environment Working Group)

ADDRESS: Department of Computer Science
Florida State University
Tallahassee, FL 32306-4019

TELEPHONE: (904) 644-5452
E-mail: (ARPAnet) tbaker@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 3., 6., 7., 9., 10., 12.

PROBLEM:

Real-time applications require that programs (and tasks) be able to start up more quickly than is possible if all elaboration is performed at execution time. A well-defined class of constructs, defined similarly to static expressions but larger in scope, is needed; such that a programmer can rely on the elaboration of these constructs not taking any execution time. We call constructs whose elaboration does not take any execution time "pre-elaborated".

SPECIFIC REQUIREMENT/SOLUTION CRITERIA:

There should be a standard class of constructs that are guaranteed able to be pre-elaborated. We call this class the "standard pre-elaboratable constructs".

There should be a way for a programmer to require that all constructs in this class be pre-elaborated. A compiler need not be required to support pre-elaboration, since this feature may not be needed by all Ada applications, but a request for pre-elaboration should not be ignorable. That is, it should result in failure of the compilation, as with other implementation limitations.

Like static expressions, this class would impose a minimum requirement on all compilers that support pre-elaboration, and would not limit any compiler from performing further optimizations. The standard pre-elaboratable constructs should include simple forms of aggregate objects, subtypes, subprograms, tasks, and packages.

Pre-elaborated objects should be ROM-able.

IMPORTANCE:

Applications requiring this feature will not use Ada, or will rely on features of a specific (possibly custom) compiler. This feature requires a language change for two reasons: (1) conformance cannot be strongly checked (meaning a program cannot be rejected for mis-use of the pragma or because the compiler does not support the feature) unless the pragma is language-defined; (2) re-use and portability of programmers, programming styles, and program components will rely on some minimum level of agreement about this feature across compilers.

CURRENT WORKAROUNDS:

This ARR addresses a kind of optimization, one that is sufficiently critical for real-time applications that some standard support seems necessary. In an application whose design is driven by timing constraints, it is necessary to know a-priori which constructs are "safe" with regard to not imposing any significant run-time costs. It is typical of many real-time applications that the entire application, or certain tasks within it, must be able started (and restarted) instantly, more or less. This implies that constructs which impose run-time elaboration costs be avoided. It is therefore important that application programmers and "real-time" Ada compilers agree upon a set of constraints, so that if a programmer respects these constraints and requests pre-elaboration, the compiler can be relied upon not to generate any significant amount of run-time elaboration code.

While it is clear that each compiler must set a limit on how far to go in compile-time elaboration. In the absence of a standard this limit will vary widely. The absence of such a standard makes it impractical to develop a common style of real-time programming, and difficult to re-use real-time software components across compilers. Any application concerned about start-up time will need to define its own set of usage restrictions to fit a particular compiler, and may even require a custom compiler.

POSSIBLE SOLUTIONS:

The present Ada standard sets a precedent by defining a class of "static" expressions, which a compiler is supposed to be able to evaluate at compile-time. This class is more narrowly defined than necessary, but it is sufficient to be useful. The class of pre-elaboratable constructs proposed here would be a generalization of this idea to more complex constructs, including certain simple forms of aggregate objects, subtypes, procedures, packages, and tasks. The class would be sufficiently narrow to insure the constructs can be elaboratable at compile time, but sufficiently large to permit programming a useful range of real-time applications.

By giving this class a name and making its definition standard, real-time programmers are given a framework within which they know it is "safe" to operate, given they use a compiler that supports this feature. By requiring compilers that do not support this feature to reject programs that request it, programmers will know that programs that compile are "safe".

In order to illustrate what we mean, and provide a starting point for discussion, we will define a class of constructs that might be pre-elaboratable. The intent is that a programmer might request that certain library packages be pre-elaborated, by means of a pragma. The pragma would only be allowed for library packages that contain only "allowable" components, where "allowable" is specifically defined below.

- A static expression is allowable.
- A scalar type with a static constraint is allowable.
- A subtype is allowable if its base type is allowable, the constraint is static, and elaboration of the subtype raises no exceptions.
- An array type is allowable if the component and index subtypes are allowable.
- A record type is allowable if every component is of an allowable subtype.
- An access type is allowable if the designated subtype is allowable.
- An allocator is allowable if its subtype and initial value (if any) are allowable.

- An aggregate is allowable if its subtype, component values, and any expressions used as array index choices are allowable.
- An object declaration is allowable if its subtype is allowable and is unconstrained or has an allowable constraint, and the declaration is either
 - a variable declaration and has no initial value, or
 - a constant declaration and the value is specified by an allowable expression.
- A type or subtype declaration is allowable if the type or subtype is allowable.
- A subprogram declaration is allowable if ELABORATION_CHECK is suppressed, or if no separate specification is given for the subprogram.
(Note: Subprogram declarations do not include generic units and generic instantiations.)
- A generic instantiation is allowable if the generic unit, its body, and all generic parameters are allowable, and the instantiation does not raise any exception.
- A package is allowable if all its component declarations are allowable and its body has no sequence of statements.
- A task (type) is allowable if all the declarations in its declarative part are allowable and it does not occur within another task or subprogram body.
- A generic declaration is allowable if all expressions and types in its formal part are allowable, and its body is a subprogram or allowable package body.

Only these declarations, expressions, types, and subtypes are allowable.

The requirements defined above are a compromise between ease of implementation and generality. They could be somewhat relaxed, at a price. For example, initial values could be allowed in variable declarations, but then restarting a program quickly might be difficult. Similarly, expressions involving selection of components and slices of allowable array and record constants might be allowed, but the benefit was felt to outweigh the extra cost. Even though certain kinds of nested tasks could conceivably be pre-elaborated, we have ruled them out for similar reasons. We intend to have ruled out all constructs whose elaboration might require the execution of a statement. In contrast, allocators are permitted in expressions defining the values of constants, even though supporting this may not be easy. This is because access types are considered a necessity for real-time programs.

DIFFICULTIES TO BE CONSIDERED:

Compilers which support this feature will be more complex.

REFERENCES/SUPPORTING MATERIAL:

ARTEWG CIFO

INTRODUCE INHERITANCE INTO ADA

DATE: June 12, 1989

NAME: Jurgen F H Winkler

ADDRESS: Siemens AG ZFE F2 SOF3
Otto-Hahn-Ring 6
D-8000 Munchen 83
Fed Rep of Germany

TELEPHONE: +49 89 636 2173

ANSI/MIL-STD-1815A REFERENCE: 3., 7.

PROBLEM:

Ada provides some elements of object oriented programming (OOP) but it lacks the mechanism of inheritance, which is very useful mechanism, and which can save a lot of recompilation and retesting effort.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

No good workarounds known. See Ada Letters (Vol. III, No. 5 (1988) p 37-46) for a possible workaround.

POSSIBLE SOLUTIONS:**SOLUTION 1**

Inheritance can be provided along the following lines:

- a) Package type (analogous to task type)
- b) Inheritance clause (subtype)

This leads to the following additional rule for package specification:

```
package_specification ::=  
    package type identifier [subtype of package_name] is  
        {basic_declarative_item}  
    [private  
        {basic_declarative_item}]  
    end [package_simple_name]
```

The semantics are as follows:

- a) Package types are private types.
- b) The declarative region of the package type designated by "identifier" (the subtype) is logically contained immediately in the declarative region of the type designated by "package_name" (the

supertype) at the point after the last declaration immediately contained in the supertype. The package type designated by "identifier" is called a direct subtype of the package type designated by "package_name".

A package type S is called a subtype of a package type T iff

- c1) $S = T$, or
 - c2) S is a direct subtype of a subtype of T.
- c) An object of the subtype consists of the data objects of the supertype (inheritance) followed by those declared in the subtype. Each object of a package type has its own set of data components.
 - d) All subprograms declared in the public part of the supertype are implicitly declared in the public part of the subtype. (Inheritance).
 - e) If OT is an object of package type T and S a subprogram declared in the public part of T the construct "OT.S(...)" means the invocation of S in the context of the data components of OT.
 - f) Assignment compatibility for package types:
the type of the rhs must be a subtype of the type of the lhs, and assignment means projection.
 - g) Assignment compatibility for access types designating package types:
the type of the object designated by the rhs must be a subtype of the designated type of the lhs.
This may require a dynamic check.
Assignment means copying of the access value.
A consequence of this rule is that an access variable may designate objects of any subtype of its designated type.

REMARKS:

- R1) The term "object" is used here in the Ada sense.
- R2) The term "package subtype" could be replaced with e.g. "type extension" [Wir 88], because there are differences between the package subtypes and other subtypes in Ada.
- R3) The proposal does not deal with virtual subprograms. They can be included also. The proposal is intended to give the flavor of an object-oriented extension of Ada, but not to define all details of such an extension.

SOLUTION 2

A combination of Wirth's type extension [Wir 88] and Ada's derived types would also introduce some form of inheritance into Ada. The solution would be limited solution because the type extension proposal of Wirth works for record types only.

REFERENCES:

[Wir 88] Wirth, N.: Type Extensions, TOPLAS 10,2 (1988) 204..214

FINALIZATION

DATE: April 4, 1989

NAME: C. Lester (from information supplied by members of Ada-UK)

ADDRESS: School of Information Science
Portsmouth Polytechnic
Portsmouth, PO1 2EG
England

TELEPHONE: +44-705-598943 after 1 pm EST

ANSI/MIL-STD-1815A REFERENCE: 3.2, 3.7, 7.1, 10.5

PROBLEM:

Ada provides adequate facilities for the initialization of variables (3.2), record components (3.7), and packages (7.1): in the case of library packages great care is taken to ensure that the initializations occur in an acceptable order (10.5).

However, for everything that might need initializing, there are likely to be symmetric needs for finalization - and Ada provides no facilities for that (unlike Simula, which was a major formative influence on these aspects of Ada - see section 1.3 of the LRM: also unlike the draft Extended Pascal, which shows the Pascal community's awareness of the need for finalization to match the initialization facilities being introduced in Extended Pascal).

Examples:

- A "counter" variable, initialized to zero, and which should return to zero before exit from the declaring scope - there is currently no easy means to determine that it returns to that value;
- a file which needs opening in a package initialization part - there is currently no easy means to close it (perhaps after first checking that it was left at end-of-file);
- ADTs supplied by packages, and which should arrive in some final state before being extinguished;
- complements of anything else for which a package initialization part could be used.

IMPORTANCE:

Depends on one's point of view:

- **ESSENTIAL** to the guy habituated to it in other languages;
- **ESSENTIAL** to prevent Ada from being the subject of ribaldry over the obvious inconsistency;
- **MILDLY DESIRABLE** to the guy numbered to the perpetual irritation of the workarounds;
- **IRRELEVANT** to the guy in the habit of writing un-robust code.

That Ada should be able to do anything that any other language can do is (of course) a nonsense argument on its own, but in this case SIMULA experience strongly suggests that such facilities are highly desirable for all the reasons that SIMULA-like initialization facilities were included in Ada.

CURRENT WORKAROUNDS:

Mirror-imaging the sort of messy code that the initialization facilities were supposed to supplant, and with all the disadvantages of the supposedly supplanted messy code.

POSSIBLE SOLUTIONS:

For packages, a SIMULA-like solution:

```
package body CLEANS_UP_AFTER_ITSELF is
...
begin
...          -- initialization code
terminate    -- very Ada-like to reuse a keyword!
...          -- finalization code
end CLEANS_UP_AFTER_ITSELF;
```

Superficially a minor extension, it has catches:

- from the point of view of the definition, it needs a new concept of "de-elaboration" in the inverse order of elaboration (suppose something declares two package bodies, A then B:
 - A initializes by opening a file;
 - B initializes by asking A to do input;
 - ...
 - B finalizes by asking A to be sure it got to end-of-file;
 - A finalizes by closing the file;
 any other order would be hard to live with in that and other scenarios). De-elaborations would need to be implied by the end matching the begin following the declarative part declaring the self-finalizing things-these "implicit things happening at an end" are also a new definitional concept (except for the funny things that happen at an end matching a begin following declarations of tasks). In the case of self-finalizing library units, the finalizations would need to come after main program termination subject to the inverse of the current partial ordering of library unit elaborations: one option is that the finalizations be in exactly the inverse order of the chosen total ordering of library unit elaborations.
- interaction with termination of tasks declared in the same declarative sequence as the package body, or (if the package body is a compilation unit) with library tasks - would the finalizations precede the required termination of the tasks, follow it, or would it be a case of 'order not defined by the language'?
- interaction with exceptions raised by the sequence of statements preceding the 'terminate' - would we permit exception handlers before terminate? If there were not such a handler (whether allowed or not), what would be the propagation rules?

For variables: there is not one most-obvious solution. Here are two candidates for consideration:

- decree that such variables ought be declared only in packages, and pass the buck to the package's finalization part (this is Extended Pascal's kludge).
- some way of specifying a test that the value of the variable must pass before de-allocation of the variable (messy to add to the definition, for two reasons: 1/ it would be radically novel syntax; 2/ then current definition never clearly says that variables can cease to exist, when, how, why, etc., so that would need clarifying first!); if the test fails then something has to happen (also messy, because it's not obvious what should happen - PROGRAM_ERROR? if so, where would it be considered to be raised?)

For record components: this is now finalization of all variables of given (record) type, so the Extended Pascal kludge becomes very dubious. The "specified test" becomes attractive, but is yet another case in which something is executed in a scope very different from where it is defined - the rules would presumably be like those for initializations expression itself.

These issues were ducked in the original design for lack of time to consider them more fully. We now have the time (even if it takes till 200X).

IDENTIFIER LISTS AND THE EQUIVALENCE OF SINGLE AND MULTIPLE DECLARATIONS

DATE: March 22, 1989

NAME: E.N. Thomas

DISCLAIMER: The views expressed in this note are those of the author, and do not necessarily represent those of SD-Scicon PLC.

ADDRESS: SD-Scicon PLC
Pembroke House
Pembroke Broadway
CAMBERLEY
Surrey
UK
GU15 3XD

TELEPHONE: +44 276 686200

ANSI/MIL-STD-1815A REFERENCE: 3.2(10), 3.2(11), 3.2(12), 3.8.1, 7.4, 6.3.1

PROBLEM:

Section 3.2 defines an equivalence for use throughout the reference manual for single and multiple object declarations. Object declarations, however, are not the only place where `identifier_list` is used, and 3.2 extends the equivalence to number declarations, component declarations, discriminant specifications, parameter specifications, generic parameter declarations, exception declarations, and deferred constant declarations.

Unfortunately, this equivalence is not universally accepted in the rest of the manual, and further the ability to use lists (`identifier_list` or equivalent) is not provided for some forms of declaration.

Non-equivalences

Although 3.2 defines the equivalence for component declarations and parameter specifications, 6.3.1 does not accept this in defining conformance of subprogram specifications and discriminant parts (3.8.1(4) and 7.4.1(3)). This leads to the ridiculous example in 6.3.1(8), which should not cause any problems at all.

The equivalence rules in 3.2(10) allow for "whatever appears in the right of the colon in the multiple object declaration", and define the effective order of the equivalent single declarations. In the case of deferred constant declarations, 7.4.3 has no difficulty allowing any combination of multiple and single object declarations for the deferred and full declarations. A further ridiculous result of 6.3.1 is the purposes of overloading, yet to not conform.

Missing lists

Several forms of declaration do not provide the ability to use lists, although an extended form of 3.2 would give no semantic difficulty in interpreting their meaning. In some cases this is equivalent to identifier list, but in others the missing construct is composite. These include:

- Incomplete type declarations, 3.8.1
- Private type declarations, 7.4.
- Multiple objects of the same (sub)type initialized to (textually) differing values.

IMPORTANCE: ESSENTIAL

Regularity is compromised without these, and the design goals of 1.3(3) are violated ("a small number of underlying concepts integrated in a consistent and systematic way").

CURRENT WORKAROUNDS:

None for the missing consistency caused by non-equivalences, but verbose forms of declaration cover the missing lists -- for example package ASCII in C(15), and constructs like the following.

```
type A;
type B;
type C;
type D;
```

Instead of the following.

```
type A, B, C, D;
```

The package ASCII case is an example of verbosity introducing hazards for program maintenance -- the constant declarations are not able to encapsulate the fact that they are all of the same type other than by repeating the "constant CHARACTER" part (the author of the reference manual even gave up in the case LC_A to LC-Z, and used" ...!").

POSSIBLE SOLUTIONS:**Non-equivalences**

Apply the concepts of section 3.2 consistently to the places discussed above.

Missing lists

Extend the syntax as follows for type declarations. The new rule `type_signature` could also be used in existing syntax rules for consistency, such as `full_type_declaration` in 3.3.1(2).

```
private_type_declaration ::=
    type_signature_list is (limited) private

incomplete_type_declaration ::=
    type_signature_list;

type_signature_list ::=
    type_signature {, type_signature}

type_signature ::=
```

identifier {discriminant_part}

For multiple object initialization, rather more departure is required from the existing style of object declaration in Ada, but this really arises from Ada's failure to conform to established style in the first place, (c.f. Algol's

```
INTEGER a, b, c;
```

or FORTRAN's

```
INTEGER A, B, C;
```

).

The syntax needs to associate closely each identifier and its initializing expression, and avoid expressing them as two lists that need to be matched one to one. Thus it needs to have a list of identifier_expression pairs. The following syntax is suggested, being constructed out of parts of 3.2, and preserving the observed fact that nowhere in the existing syntax are two adjacent non-reserved identifiers allowed. The number declaration alternative seems hardly justified, but is included for completeness.

```
object_declaration ::=
    <<as at present followed by>>
    | [constant] subtype_indication declare initialize_object_list;

number_declaration ::=
    <<as at present followed by>>
    | constant declare initialize_number_list;

initialize_number_list ::=
    initialize_number {, initialize_number}

initialize_number ::=
    identifier := universal_static_expression

initialize_object_list ::=
    initialize_object {, initialize_object}

initialize_object ::=
    identifier := expression
```

Examples using this syntax are:

```
constant CHARACTER declare
    LCA_A := 'a',
    LCA_B := 'b',
    LCA_C := 'c',
    LCA_D := 'd',

constant declare PI := 3.1415926536,
    MAX := 500,
    POWER_16 := 2:1E16;

INTEGER declare COUNT := 0, SUM := -1;
```

Semantic rules will be needed as for other declarations concerning use of an identifier before the end of the declaration, and whether or not multiple constant declarations like the one given above can be of mixed universal real and universal integer.

CONSTANTS CANNOT USE DEFAULT VALUES

DATE: March 22, 1989

NAME: E.N. Thomas

DISCLAIMER: The views expressed in this note are those of the author, and do not necessarily represent those of SD-Scicon PLC.

ADDRESS: SD-Scicon PLC
Pembroke House
Pembroke Broadway
CAMBERLEY
Surrey
UK
GU15 3XD

TELEPHONE: +44 276 686200

ANSI/MIL-STD-1815A REFERENCE: 3.2.1(2), 3.3(6).

PROBLEM:

When a variable is declared, it is able to make use of any initial default value for the relevant type. However, when a constant is declared it cannot, because 3.2.1 requires explicit initialization.

This is particularly a problem in the case of a private type from another package, because it is quite possible that no suitable value is visible for declaring constants. The properties described for the operations on the private type may be such as to make it clear that a default must be present, even though the private declaration is not available to the user. For example, in the case of the limited private type `FILE_TYPE` in the predefined package `DIRECT_IO`, it is clear that it must have a default initial value because of the properties of the defined operations like `OPEN`, `RESET`, `CLOSE`, and so on, and because there is no operation to initialize a newly declared object of `FILE_TYPE` ready for use by the other operations-the file is described as initially closed, not undefined. Of course in the case of a limited private type no user constants can be declared anyway, but the ability to deduce a default value is equally valid for nonlimited types.)

Although it is most noticeable for private types, the same applies to variable of any type with an initial value.

IMPORTANCE: ESSENTIAL

Regularity is compromised without these, and the design goals of 1.3(3) are violated ("a small number of underlying concepts integrated in a consistent and systematic way").

CURRENT WORKAROUNDS:

One possibility at present is to declare a dummy variable, from which the value for the constant can be obtained, as in the following (contrived) example. The presence of such dummy variables is a maintenance hazard.

```
type POINT is access INTEGER;  
...  
DUMMY : POINT -- Has value null  
NULL_POINT : constant POINT := DUMMY;
```

POSSIBLE SOLUTIONS:

The syntax for default expressions used for object declarations, component declarations, and parameter specifications should be extended to permit an explicit "default". In order to avoid the introduction of a new reserved word, it is suggested that the box compound delimiter of 2.2(6) should be used, as has been suggested in another Ada9X Revision Request relating to explicit defaults for actual parameters at subprogram calls.

MAKE "EXCEPTION" A PREDEFINED TYPE**DATE:** January 19, 1989**NAME:** Damon Lease**ADDRESS:** Strategic Electronic Defense Division
GTE Government Systems Corporation
100 Ferguson Drive
P.O. Box 7188
Mountain View, CA 94039**TELEPHONE:** (415) 966-3439**ANSI/MIL-STD-1815A REFERENCE:** 3.3.1, 11.1, 11.2**PROBLEM:**

In order to adequately handle all possible exceptions within any block of code, it is necessary, at times, to explicitly handle many more exceptions in a handler than is desirable. This leads to poor programming techniques, such as overuse of the "when others" handler. When this occurs many times in a large system, traceability suffers greatly. During integration (CSC, CSCI, SYSTEM) and test, this can become extremely expensive when untraceable errors occur.

IMPORTANCE:

The importance of this depends on individual programming styles. Personally, for code I write, I consider it to be low, maybe a two or three. This is because I make it a point to explicitly handle a large set of exceptions, and rarely use a "when others" handler. However, for other programmers, this may seem more important. Overall, I would probably rate the possible solution below a five or six. If no change is implemented, programmers will continue to abuse the "when others" handler in the interest of getting to the next piece of code.

CURRENT WORKAROUNDS:

Explicitly handle all exceptions that could possibly propagate through any exception handler. Always include the optional exception handler.

POSSIBLE SOLUTION:

Modify the language so that the reserved word "exception" is a predefined type. Modify the syntax and semantics of exception handlers.

Specific suggestions:

Create a new kind of type definition known as an `exception_type_definition`. This will be added to the `type_definition` in section 3.3.1 of ANSI/MIL-STD-1815A. Define `exception_type_definition` as follows:

```
exception_type_definition ::= exception
```

This would cause the definition of an `exception_type` to be as follows:

```
type my_exception_type is exception;
```

Also allow subtypes of exception types and subtypes. That is, the following statement would be legal, and create an exception subtype of my_exception_type:

```
subtype my_other_exception_type is my_exception_type;
```

This will allow the creation of hierarchical classes of exceptions.

Allow exceptions to be declared from any exception type or subtype, including the original exception type. That is: an_exception: exception; my_exception: my_exception_type; would both be legal.

Therefore, the syntax of the exception declaration would be:

```
exception_declaration ::= identifier_list : subtype_indication;
```

The predefined exceptions would be of type predefined_exception. This implies that predefined_exception is an exception type. Exceptions cannot be declared of type predefined_exception.

To coincide with this, the syntax of the exception handler would change.

The exception handler would allow exception handlers to handle exceptions as individual exceptions, as classes of exceptions (those of the same type or subtype), or with a "when others" handler. The syntax will change so that an exception_choice will be:

```
exception_choice ::= exception_name |  
                    in exception_type_mark |  
                    others
```

It will still remain true that an individual exception may only be explicitly handled once. In particular, as a consequence of allowing subtypes of exception types, it is possible that a single exception could be covered by multiple "in exception_type_mark" handlers. This will not be permitted by the compiler. If exception my_exception has been declared as above, the following handler will be flagged as illegal by the compiler: exception when my_exception => sequence_of_statements; when in my_exception_type => sequence_of_statements; end simple_name;

Also, the following will be illegal:

```
procedure p is
```

```
type ex1_type is exception;
```

```
subtype ex2_type is ex1_type;
```

```
ex1: ex1_type;
```

```
ex2: ex2_type;
```

```
begin

    sequence_of_statements;

exception

when in ex1_type =>

    sequence_of_statements;

when in ex2_type =>

    sequence_of_statements;

end p;
```

The above two examples are illegal because two different handlers explicitly handle the same exception.

The proposed changes shown here can be easily implemented for a few reasons. They are:

- No new reserved words would be added to the language.
- All existing code would still compile and execute the same as it does now because the reserved word exception is still being allowed to declare an exception.
- The semantic checking of the handler would not be much more difficult because a mechanism is already in place to ensure that no exception is explicitly handled twice.

INITIALIZATION FOR NONLIMITED TYPES**DATE:** May 23, 1989**NAME:** Jurgen F H Winkler**ADDRESS:** Siemens AG ZFE F2 SOF3
Otto-Hahn-Ring 6
D-8000 Munchen 83
Fed Rep of Germany**TELEPHONE:** +49 89 636 2173**ANSI/MIL-STD-1815A REFERENCE:** 3.3.1**PROBLEM:**

Allow initialization for all nonlimited types.

Ada allows the initialization for record types only (for the components not for the type itself), and it defines a default initialization for access types. This irregular structure of the language complicates the learning of the language, and complicates the formulation of correct and robust programs.

IMPORTANCE: IMPORTANT**CURRENT WORKAROUNDS:**

Provide the initialization in the variable declaration or by an assignment statement.

POSSIBLE SOLUTIONS:

```
full_type_declaration ::=  
    type identifier [discriminant_part] is type_definition [:=expression];
```

NON-CONTIGUOUS SUBTYPES OF ENUMERATION TYPES**DATE:** April 4, 1989**NAME:** Eric F. Heck**ADDRESS:** GE Aerospace GCSD
Mail Stop 13-7-5
Camden, NJ 08102**TELEPHONE:** (609) 338-4776**ANSI/MIL-STD-1815A REFERENCE:** 3.3.2(2)**PROBLEM:**

When interfacing Ada code to external devices the use of enumeration types, with representations clauses, is used. Because values are defined for the enumeration type, via a representation clause, an order for the elements is implied. In the Ada code, however, subtypes of the enumeration type may need to be declared. The problem is that these subtypes may need to have non-contiguous elements from the base type. For example:

```
type Network_Type is
    (UHF_High, IWC, UHF_Low, VHF_High, VHF_Low);

for Network_Type use (UHF_High      => 16#02#,
                      IWC           => 16#04#,
                      UHF_Low       => 16#10#,
                      VHF_High      => 16#12#,
                      VHF_Low       => 16#14#.);
```

Network : Network_Type

An embedded real-time system may need the following subtype:

```
subtype Hopping_Subtype is Network_Type
    (UHF_High, UHF_Low, VHF_Low);
```

Currently, Ada does not allow the subtype declaration for Hopping_Subtype as defined above.

IMPORTANCE: IMPORTANT

This request is important to those trying to use Ada on a system that interfaces with external devices. If this request is not incorporated into the Ada revision, those types of systems could still use Ada; however, they may choose to declare objects as bytes instead of using enumeration types with representation clauses. This would make the code look more like Fortran than Ada.

CURRENT WORKAROUNDS:

- 1) Declare objects to be of Byte_Type (range 0..255) instead of using enumeration types. Thus instead of coding:

If Network = UHF_High

we would code:

If Network = 16#02#

This produces poor Ada code!!!

- 2) Declare a new enumeration type that contains the values of the subtype declared above. The problem here is that we then must explicitly convert from one type to the other type.

POSSIBLE SOLUTIONS:

Implement the reserved area for types and subtypes as linked lists.

VARIABLE FINALIZATION

DATE: December 2, 1988

NAME: James L. Allison

ADDRESS: Lockheed Missiles and Space Company
1150 Gemini Avenue
Houston, TX 77058

TELEPHONE: (713) 282-6498

ANSI/MIL-STD-1815A REFERENCE: 3.3(6), 8.2

PROBLEM:

It is often necessary to define a type that must have some processing done before it can be released. There are several reasons for this. It may be necessary to keep track of how many variables of a particular type exist at any given time. The variable may contain a 'key' that allows access to a resource. The variable may contain a heap pointer. As far as I can tell, this problem only arises with limited types. With any other type multiple copies can exist with NO chance of correcting this.

The case where the variable contains a heap pointer is probably the most common.

For example:

```
with heap_manager;  
procedure do_something is  
    temp:heap_manager.limited_type_with_hidden_pointer;  
begin  
    ...  
    heap_manager.get_space(temp);  
    ...  
    ...  
    Heap_manager.release_space(temp);  
    ..  
end;
```

If the application programmer forgets to make the call to release the space, that space can be lost forever. Each entry into this procedure locks up more space, with no hint that anything is wrong. In general, no garbage collection procedure could ever detect that locked up space. This could cause a system failure hours or months in the future.

IMPORTANCE: ESSENTIAL

This is a hole in the firewall between packages; every variable of a type that must be finalized is a potential trouble spot, waiting only for an application programmer to forget the cleanup call. The compiler does not now have the information to check for this oversight.

CURRENT WORKAROUNDS:

Provide a cleanup procedure with every type that needs it, and instruct the programmers that they must use it. There is no way to enforce this use, other than an a manual audit.

It is also possible that in a large project neither the application programmer nor his supervisor understand the importance of variable finalization.

POSSIBLE SOLUTIONS:

Allow a (limited private?) type definition to specify a cleanup procedure to be invoked automatically before a variable of that type is freed. The forced finalization must be on the type, not on the variable declaration. It would be just as easy for an application programmer to forget to specify cleanup if it were on the variable.

The spec could be as follows, with the suggested change indicated.

```
package heap_manager is
  ...
  type limited_type_with_hidden_pointer is limited private;
  ...
  ...
  private
    procedure release_space(x:in out limited_type_with_hidden_pointer);
    type limited_type_with_hidden_pointer => release_space is record
      ...
      ...
    end record;
end heap_manager;
```

It is debatable whether or not to allow export of the cleanup procedure, or even whether or not to allow the argument list to be provided. The appearance of a name in the type definition might be all that is needed.

This solution would not introduce any new tokens or reserved words, and the impact to existing software would be confined to those packages that really need the forced finalization.

VISIBILITY OF BASIC OPERATIONS ON A TYPE

DATE: December 5, 1988

NAME: David Brookman

ADDRESS: Magnovox Electronic Systems Company
1313 Production Road
Department 542
Fort Wayne, IN 46808

TELEPHONE: (219) 429-4440
E-mail: CONTR22ONOSC-TECR.Arpa

ANSI/MIL-STD-1815A REFERENCE: 3.3.3, 8.1(10), 8.3(7)

PROBLEM:

Basic operations on a type, defined in a package specification, are not directly visible in another module which imports that package. If the basic operations are defined as infix operations, such as "=", then the infix notation cannot be used in the other module.

IMPORTANCE: ADMINISTRATIVE

These forces the programmer to use one of the three workarounds listed below.

CURRENT WORKAROUNDS:

1. Place a "Use" clause in the module importing the package. This ruins the traceability of identifiers. Therefore this is a very poor workaround.
2. Use the fully qualified form of the operations as a prefix operation. This reduces the readability of the resulting code.
3. Rename the operation so that it becomes directly visible. This is the best workaround, but it still is not ideal. It forces ugly rename clauses and may cause errors. If "<" is renamed to ">", the error could be quite difficult to detect.

POSSIBLE SOLUTIONS:

1. Make the basic operations directly visible.
2. Provide a "half-use" clause that only makes basic operations directly visible.

MULTIPLE TYPE DERIVATIONS**DATE:** March 17, 1989**NAME:** Mats Weber**ADDRESS:** Swiss Federal Institute of Technology
EPFL DI LITH
1015 Lausanne
Switzerland**TELEPHONE:** 0041 21 693 11
E-mail: madmats@elma.epfl.ch**ANSI/MIL-STD-1815A REFERENCE:** 3.4**PROBLEM:**

The type derivation mechanism has undesirable effects when multiple types declared in the same package visible part are derived. For example: package Graph_Handler is type Vertex is private; type Arc is private; function New_Arc (Initial, Final : Vertex) return Arc; function Initial (Of_Arc : Arc) return Vertex; function Final (Of_Arc : Arc) return Vertex; private

```
...  
end Graph_Handler;  
  
...  
  
type Node is new Graph_Handler.Vertex;  
-- derives subprograms  
-- function New_Arc (Initial, Final : Node) return Graph_Handler.Arc;  
-- function Initial (Of_Arc : Graph_Handler.Arc) return Node;  
-- function Final (Of_Arc : Graph_Handler.Arc) return Node;  
type Edge is new Graph_Handler.Arc;  
-- derives subprograms  
-- function New_Arc (Initial, Final : Graph_Handler.Vertex) return Edge;  
-- function Initial (Of_Arc : Edge) return Graph_Handler.Vertex;  
-- function Final (Of_Arc : Edge) return Graph_Handler.Vertex;  
-- What we need is  
-- function New_Arc (Initial, Final : Node) return Edge;  
-- function Initial (Of_Arc : Edge) return Node;  
-- function Final (Of_Arc : Edge) return Node;
```

IMPORTANCE: IMPORTANT**CURRENT WORKAROUNDS:**

Use a lot of type conversions.

POSSIBLE SOLUTIONS:

Add multiple type derivations to the language, with a syntax like type (Node, Edge) is new (Graph_Handler.Vertex, Graph_Handler.Arc); which would derive the desired subprograms.

DERIVED TYPES

DATE: April 21, 1989

NAME: Stewart French

ADDRESS: Texas Instruments
P.O. Box 869305 MS 8435
Plano, TX 75086

TELEPHONE: (214) 575-3522

ANSI/MIL-STD-1815A REFERENCE: 3.4(17)

PROBLEM:

Current use of derived types is clumsy; current syntax requires renaming of derived types and operations.

IMPORTANCE: IMPORTANT

CONSEQUENCES:

Implementors will avoid use of derived types due to verbose declaration requirements of derived types.

CURRENT WORKAROUNDS:

Avoidance of construct.

POSSIBLE SOLUTIONS:

OVERFLOW AND TYPE CONVERSION**DATE:** February 21, 1989**NAME:** SY Wong**ADDRESS:** 5200 Topeka Drive
Tarzana, CA 91356**TELEPHONE:** (818) 345-6274
E-mail: hermix!sywong@rand-unix.arpa**ANSI/MIL-STD-1815a REFERENCE:** 3.5.4, 4.5.3, 4.5.5, 4.6-1**PROBLEM:**

Integer multiplication is capable of generating a double size word, and most machines are implemented as such and also uses the same facility to accommodate a division remainder. The Ada definition of one size for the multiplier, multiplicand and product would result in overflow for most of the cases. In general, $(A/B)*B \neq A$ in the Ada definition.

The same problem even manifests itself in additive operations:

x: integer := A + B + C + D;

may overflow in intermediate results even x has been analyzed to yield an in-range integer. A more precise definition of overflow and type conversion is required.

IMPORTANCE: ESSENTIAL**CURRENT WORKAROUNDS:**

Use double_length integers and wastes performance and resources.

To require explicit conversion of expression to long_integers and then convert back to integer is also extremely awkward and not always necessary. First converting to long_integer is not the same as integer arithmetics with long_integer intermediate results.

POSSIBLE SOLUTIONS:

A compromise might be to require/allow double_length intermediate results, consistent with arithmetic basics and arithmetic unit designs. Overflow exception is raised only on the final result or when intermediate double_length results are exceeded.

a, b: integer;

long_integer (a * b) or long_integer (a + b) should never raise overflow exception.

UNSIGNED INTEGERS

DATE: June 27, 1989

NAME: Neil Salant

ADDRESS: ITT Avionics
Dept. 73813
390 Washington Ave.
Nutley, NJ 07110

TELEPHONE: (201) 284-3896

ANSI/MIL-STD-1815A REFERENCE: 3.5.4

PROBLEM:

Ada does not provide unsigned integer types which allow the "sign bit" to be interpreted as a part of the absolute numeric value of an integer object. Under the current language definition, it is impossible to represent the numeric values $0..(2^{**n})-1$ in an n bit integer, due to the Ada requirement that integer types be symmetric about zero, i.e. have a sign bit.

IMPORTANCE: ESSENTIAL

Unsigned integer representation is needed to interface with various hardware devices as well as non-Ada software systems which make use of such fields. Unsigned integers are additionally needed for implementation of efficient overflow counters.

CURRENT WORKAROUNDS:

Signed integers can be used for values between 0 and $(2^{**n}-1)$, where n is the number of bits in the underlying representation. However, operations that cause the sign bit to go from logical 0 to 1 cause an overflow, which should not be the case for an unsigned integer with range $0..(2^{**n})-1$. It is possible to try to increase the value of n by type conversion to a "longer" integer type, but here various implementation dependencies and inefficiencies are encountered such as sign extension, availability of longer types, restrictions on unchecked conversion, and insertion of compiler generated temporary variables.

POSSIBLE SOLUTIONS:

Introduce a new predefined integer type to the language definition which is an unsigned integer with values of $0..(2^{**n})-1$ where n is the number of bits in the underlying representation. An overflow (value $\geq 2^{**n}$) should raise an exception (Numeric_Error or Constraint_Error); however overflow checking should be suppressible.

LIMITATION ON RANGE OF INTEGER TYPES**DATE:** May 30, 1989**NAME:** Donald R. Clarson**ADDRESS:** Teledyne Brown Engineering
151 Industrial Way East
Eatontown, NJ 07724**TELEPHONE:** (201) 389-7500**ANSI/MIL-STD-1815A REFERENCE:** 3.5.4(6)**PROBLEM:**

An implementation is required to accept any integer type as an array index, case statement selector, or discriminant. This makes it impractical to support integer types with a large range of values (e.g. -2^{63} .. $2^{63}-1$) since this would affect the size of control blocks for descriptors necessary to store these values (e.g. array bounds). Types with these large values would be necessary to store cardinal values such as the national debt in cents but are impractical for parameters of data structures.

IMPORTANCE: IMPORTANT

Private types which hide the numeric properties of these values do not support integer literals for values of these types. These types would also be useful as the full implementation of a private type which requires a large integer range or operations with a wide range of values before an overflow condition result.

CURRENT WORKAROUNDS:

Declare private types with arrays of predefined integer values in the full type declaration and subprogram bodies which emulate arithmetic operations for these types. Provide conversion routines from a long floating point type to allow some literal values beyond the range of the largest predefined integer type for the implementation.

POSSIBLE SOLUTIONS:

An implementation should be allowed to accept integer type definitions with a range beyond a practical value for discrete types and then reject the use of those types for array indices, loop parameters, case statement selectors or discriminants.

CATENATION OPERATION FOR ONE-DIMENSIONAL CONSTRAINED ARRAYS

DATE: June 13, 1989

NAME: Donald R. Clarson

ADDRESS: Teledyne Brown Engineering
151 Industrial Way East
Eatontown, NJ 07724

TELEPHONE: (201) 389-7500

ANSI/MIL-STD-1815A REFERENCE: 3.5.4(6)

PROBLEM:

Catenation operation for one-dimensional constrained array declarations raise CONSTRAINT_ERROR even for intermediate values in a larger expression.

```
-- Explores catenation operation for a constrained array type
--
-- AUTHOR:
-- Donald R. Clarson
-- based on Doug Bryan, Ada Letters, Sep/Oct 1988 and May/June
-- 1989
--
with Float_TEXT_IO, TEXT_IO;
use Float_TEXT_IO, TEXT_IO;
procedure EXAMPLE_CONSTRAINED_SLICES is
  type Vector is array (1..10)
    of Float;
  V : Vector := (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0);
begin
  V := Vector ( V(6..10) & V(1..4) & 3.3 );
  -- This catenation should be an operation of the base type
  -- No CONSTRAINT_ERROR should occur
  for I in V'RANGE loop
    TEXT_IO.PUT (ITEM => "V (" & Integer'IMAGE(I) & ") --> " );
    FLOAT_TEXT_IO.PUT( ITEM => V(I),
      FORE => 5,
      AFT  => 2,
      EXP  => 0 );
    TEXT_IO.NEW_LINE;
  end loop;
  TEXT_IO.NEW_LINE;
exception
  when CONSTRAINT_ERROR =>
    TEXT_IO.PUT (ITEM => "CONSTRAINT_ERROR";
  end EXAMPLE_CONSTRAINED_SLICES;
```

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

Declare the unconstrained base type and the constrained array subtype as separate declarations.

POSSIBLE SOLUTIONS:

Change the interpretation of the rules for raising `CONSTRAINT_ERROR` in this case to delay the check until the assignment operation. This would more closely match the expected behavior of this expression.

FLOATING POINT CO-PROCESSORS**DATE:** July 13, 1989**NAME:** M. Ben-Ari**ADDRESS:** Brandeis University (until 8/89)

81 Hagedud Haivri St. (from 9/89)
Kiryat Haim 26306
Israel

TELEPHONE: 617-736-2726 or 617-332-1419 (until 8/89)
011-972-4-725905 (from 9/89)
E-mail: moti@cs.brandeis.edu (until 8/89)

ANSI/MIL-STD-1815A REFERENCE: 3.5.6**PROBLEM:**

Most microprocessors used in real-time systems implement floating point operations using a co-processor. A systems design may elect not to incorporate a co-processor for reasons of expense, space, etc. A conforming implementation should be required to support a configuration that lacks a co-processor. It should be required to support fixed point types implemented as integers. Software support for floating point is useful but possibly should not be required so that the run-time system of a hardware-constrained system can be kept small.

IMPORTANCE: IMPORTANT**CURRENT WORKAROUNDS:**

Most vendors do in fact support configurations without co-processors. Unfortunately, this may not include fixed-point operations which are important in embedded applications. Thus the use of Ada in hardware-constrained applications often leads to programming in integers rather than fixed-point.

POSSIBLE SOLUTIONS:

If the target configuration of a conforming implementation includes a separate floating point processor and the target can be sensibly run without such a processor, the implementation shall be required to support the language on the reduced configuration as well. Optional: (except for floating point types and operations).

FLOATING POINT MUST INCLUDE LONG_FLOAT AND SHORT_FLOAT**DATE:** March 21, 1989**NAME:** Joseph Fasano**ADDRESS:** 2300 Geng Road
Palo Alto, CA 94303**TELEPHONE:****ANSI/MIL-STD-1815A REFERENCE:** 3.5.7(8)**PROBLEM:**

The names of floating point types must be standardized. The fact that Long_Float and Short_Float MAY Be used offers less portability to Ada. The "MAY" should not appear in Chapter 3.

IMPORTANCE:**CURRENT WORKAROUNDS:****POSSIBLE SOLUTIONS:**

REFERENCE TO SELF IN INITIAL VALUE EXPRESSION**DATE:** May 17, 1989**NAME:** T. P. Baker**ADDRESS:** Florida State University
Department of Computer Science
207A Love Building
Tallahassee, FL 32306-4019**TELEPHONE:** (904) 644-5452
E-mail: (ARPAnet) tbaker@ajpo.sei.cmu.edu
E-mail: baker@nu.cs.fsu.edu**ANSI/MIL-STD-1815A REFERENCE:** 3.7**PROBLEM:**

There is no way to insure initialization of circular list structures, because there is no way to obtain a reference (pointer) to the structure being initialized.

IMPORTANCE: IMPORTANT**CURRENT WORKAROUNDS:**

All operations must check for uninitialized nodes. In the procedure INSERT below, this amounts to two extra if-statements.

```

package RINGS is
  type NODE;
  type LINK is access NODE;
  type NODE is
    record
      NEXT, PREV: LINK := null; -- would like to initialize to self.
      DATUM: INTEGER := 0;
    end record;
  procedure INSERT(RING, ITEM: LINK);
  ... -- more operations
end RINGS;

```

```

package body RINGS is
  procedure INSERT(RING, ITEM: LINK) is
  begin
    if RING.NEXT = null then
      RING.NEXT := RING; RING.PREV := RING;
    end if;
    if ITEM.NEXT = null then
      ITEM.NEXT := ITEM; ITEM.PREV := ITEM;
    end if;
    ITEM.NEXT := RING.NEXT; ITEM.PREV := RING;
  end;
end;

```

```
    RING.NEXT.PREV:= ITEM; RING.NEXT:= ITEM;  
end INSERT;  
...  
end RINGS;
```

POSSIBLE SOLUTIONS:

A notation could be introduced that would only be legal within an allocator, and which would return a pointer to the object currently being allocated.

One issue is choosing a notation, that does not introduce any new keywords. One solution might be an attribute of the type-mark of the access type being used.

Another issue is what to do about nested allocators. The obvious solution is to resolve such references to the allocator most immediately enclosing the reference.

This is not hard to implement, is a compatible extension of the existing semantics, and would be a convenience.

ADDITION OF ATTRIBUTES FOR RECORD TYPES

DATE: January 14, 1989

NAME: William Thomas Wolfe

ADDRESS: Home: 102 Edgewood Avenue #2
Clemson, SC 29631 USA

Office: Department of Computer Science
Clemson University
Clemson, SC 29634 USA

TELEPHONE: Home: (803) 654-7030
Office: (803) 656-2847
E-mail: wtwolfe@hubcap.clemson.edu

ANSI/MIL-STD-1815A REFERENCE: 3.7.4

PROBLEM:

The attributes applicable to a record type do not include `NUMBER_OF_FIELDS`, `FIELD_NAME` (`Field_Number`), or `FIELD` (`Field_Number`).

CONSEQUENCES:

It is not possible to construct a generic report generator which accepts an arbitrary record type and provides a procedure which, given a file of records of this type, will automatically generate a report file in which the column headers are labeled with the field names (with spaces substituted for the underscores), and the columns are automatically arranged with an appropriate layout across the page. (Assuming the availability of an appropriate set of file operations, the name of the report file would be used as the report title).

CURRENT WORKAROUNDS:

There is no easy solution. The closest it is possible to come (without accepting a very messy parameter list) is a generic report handler which accepts arbitrary procedures for the generation of page headers, column headers, etc.; this tool is more general and therefore more versatile, but does not provide the tremendous savings of time which are possible if one has the described generic report generator at one's disposal.

POSSIBLE SOLUTIONS:

Add the attributes `NUMBER_OF_FIELDS`, `FIELD_NAME` (`Field_Number`), and `FIELD` (`Field_Number`) for record types, through the insertion of three more attribute descriptions between ANSI/MIL-STD-1815A 3.7.4 (3) and ANSI/MIL-STD-1815A 3.7.4 (4).

STATIC RAGGED ARRAYS

DATE: October 23, 1988

NAME: Ken Garlington

ADDRESS: General Dynamics-Data Systems Division
P. O. Box 748
Mail Zone 5997
Fort Worth, TX 76101

TELEPHONE: (817) 762-9204

ANSI/MIL-STD-1815A REFERENCE: 3.8(8)

PROBLEM:

Certain data structures in embedded systems require the use of arrays of constant data whose components are non-homogeneous records (not all the same length). It is unclear whether the language allows the declaration of such a structure directly (some currently-validated compilers do not allow it). In addition, such a structure should require minimal data and code allocation, and should not require actions to be taken as a part of program elaboration.

SPECIFIC REQUIREMENT:

This paper makes the following assumptions:

- 1) The target architecture has an addressing mode which efficiently uses access objects which have an underlying representation of `SYSTEM.ADDRESS`.

This is true for the MIL-STD-1750A, which has "indirect" addressing modes for many of its instructions. It is also true for many other modern CISC architectures, such as the DEC VAX and Motorola 680x0 processor series.

- 2) The Ada compiler uses `SYSTEM.ADDRESS` as the underlying representation for its access types.

This is true of many (if not all) vendor's implementations of Ada cross-compilers for the MIL-STD-1750A, as well as the DEC Ada compiler. In fact, the author knows of no compilers which do not use `SYSTEM.ADDRESS` as the underlying representation. This is not surprising, given assumption #1.

There is, occasionally, a situation where the underlying `SYSTEM.ADDRESS` value refers to the address of an object descriptor, rather than the object itself. This is normally the case for arrays whose sizes are not known at compile-time (i.e., `STRING`s). This particular problem is discussed in more detail elsewhere.

- 3) There is an upper bound on the time which can be spent in the elaboration of library-level units (usually during the power-up cycle).

Full-authority flight control systems are "fail-operate" devices.

A loss of function for more than a brief interval (usually specified in the region of 20 ms) can place the aircraft into an unrecoverable condition, endangering the airframe and pilot. Usually, backup power sources are available to prevent in-flight losses of power. However, there are certain cases where a power loss of a few milliseconds can occur, requiring the program to be re-elaborated (see also #4). Therefore, it is necessary that the time spent in this process be minimized. Flight control systems (and other embedded systems) may also have maximum power-up times imposed due to mission constraints. For example, many aircraft platforms have requirements to transition to a fully-powered-down state to aircraft takeoff within a set period under alert conditions. In this situation, the time spent in program elaboration can impact the ability to meet this requirement.

- 4) Many embedded systems are not loaded from mass memory on system initiation. Rather, the program code and literal areas are stored in firmware - they are "ROMed".

This situation is often used to help meet the requirement described in #3 above. For a significantly-sized program, the time needed to load the application from an external storage device may be excessive. Therefore, only the data areas (in RAM), which are relatively small for many embedded systems, are initialized as part of the program elaboration. As a consequence of this, the RAM area must be initialized after each application of power. Therefore, the process called "pre-elaboration" (which as been suggested as an alternate solution to the run-time problems described below) is not always workable. (In addition, the "pre-elaboration" technique assumes that the elaboration sequence is insensitive to values known only at run-time. This is not always true.)

- 5) Embedded systems have limited processor power and memory available. Furthermore, a few instructions (or words) saved in a simple example can be significant when applied to a larger example.

In the workaround given, a savings of a few words for each object accessed can have great benefits when applied to a larger database. Similarly, a few instructions saved can prove important, given that the algorithm may be executed over a large number of input values.

- 6) Compilers can recognize simple optimizations and make them, but cannot recognize complex optimizations which require implicit knowledge of programmer intent.

Therefore, any proposed solution should be one which will allow the compiler to readily generate highly-efficient code and data structures without any elaboration overhead. In the example, below, a specific technique (the use of an array of "static pointers") is described. The solution should also avoid some or all of the weaknesses in the current workaround (see below).

IMPORTANCE:

Obviously, the user is given two choices: Use a "legal" (portable) Ada implementation which may not be acceptable given the available processor resources, or use an erroneous (non-portable) but efficient approach which features weak type checking and (for the naive programmer) the potential for an incorrect implementation. Neither is particularly desirable.

CURRENT WORKAROUNDS:

Databases of the type govern above are most commonly found in display applications which have data structures describing different message formats; however, other application examples include gain data tables in flight control systems and weapon data tables in fire control systems.

The following example comes from a hypothetical design document:

"In the application is a package called MESSAGE_SERVICES. Within this package is the function GET_MESSAGE which will, given an integer code ranging from 0 to 100, return a corresponding pre-defined text string of 1 to 255 characters. Strings will be of widely-varying lengths. Both the speed of execution and the memory use of GET_MESSAGE should be kept as low as possible."

1. MESSAGE_SERVICES Implementation

Here's a possible implementation of MESSAGE_SERVICES:

package MESSAGE_SERVICES is

type MESSAGE_CODE_TYPE is range 0 .. 100;

subtype MESSAGE_LENGTH_TYPE is INTEGER range 1 .. 255;

-- sure, INTEGER may not include the range 1 .. 255. And
-- my mother might be the true Queen of England.

type MESSAGE (MESSAGE_LENGTH: MESSAGE_LENGTH_TYPE) is record
VALUE : STRING(1 .. MESSAGE_LENGTH);
end record;

-- using MESSAGE type instead of STRING prohibits zero-length
-- messages and requires only one array bound (the length).

function GET_MESSAGE (MESSAGE_CODE: MESSAGE_CODE_TYPE)
return MESSAGE;

pragma INLINE (GET_MESSAGE);

end MESSAGE_SERVICES;

2. GET_MESSAGE Implementation

There are several ways GET_MESSAGE could be implemented. Each of the following sections examines a different approach. Keep in mind that the efficiency of the unit is extremely important!

2.1 Standard Programming Approaches to GET_MESSAGE

A case statement could be used to implement GET_MESSAGE, but this would probably generate at least two assembly instructions for each message, and possibly other extraneous data as well. Therefore, this would probably not be a good approach.

It turns out that array accesses on the target machine (in fact, on most target machines) can be fairly fast. Given an array, it might take as few as one or two instructions to calculate the address of a particular array element (indexed by MESSAGE_CODE) and put that address in MESSAGE. Since the instructions to do this are the same, regardless of which message is to be returned, the problem with the first approach is solved.

There are two snags with an array of text strings, however. First, not all validated Ada compilers allow different elements of an array to have different lengths. This has to do with the interpretation (and

re-interpretation) of LRM 3.6.1:16. Therefore, an array of MESSAGEs can't be declared directly. Second, if all array elements are not the same size, the array access calculation is made more complex, and the statement made in the previous paragraph may not hold. There are other approaches (i.e., making all MESSAGEs the same size or declaring an array of indices into a large string containing all MESSAGEs), but these methods are clumsy and usually not very efficient. However, there is a better alternative in Ada: the use of pointers (or access values). Consider the following example code fragment from GET_MESSAGE:

```

type MESSAGE_POINTER is access MESSAGE;

type MESSAGE_ARRAY is array (MESSAGE_CODE_TYPE) of MESSAGE_POINTER;

MSG_TABLE: constant MESSAGE_ARRAY := MESSAGE_ARRAY
  ( 0 => new MESSAGE'(2,"OK"),
    1 => new MESSAGE'(14, "SYSTEM RESTART"), ...

begin
    -- assembly equivalent is
    return MSG_TABLE(MESSAGE_CODE).all; -- LOAD MESSAGE,MSG_TABLE(MESSAGE_CODE)

end GET_MESSAGE;
```

One extra word of data per message (for the MSG_TABLE element which points to each MESSAGE) and one assembly instruction to get a MESSAGE certainly seems to be extremely fast and small! What could be wrong with this solution?

Here's what's wrong: There is a hidden piece of code which is being generated for each array element. It is hidden in the word "new." For almost all compilers, a routine must be called during the elaboration of the program to allocate each MESSAGE record in an area of memory called the "heap," returning a MESSAGE_POINTER which points to the allocated heap area. This is unacceptable from an efficiency standpoint, both because of the additional time required during system start-up and because this approach normally requires that the strings be moved from a literal area to a data area (the heap), doubling the size of the table. The "new" question, then, becomes: Is there an alternate way to generate MESSAGE_POINTERS?

2.2 A New Approach: Array of MESSAGEs Using Static Pointers

There is, in fact, a way to define a function which requires no instructions to generate a MESSAGE_POINTER. This is done by instantiating a copy of the generic UNCHECKED_CONVERSION as follows (assuming the types defined in the previous approach are defined):

```
function MAKE_POINTER is new UNCHECKED_CONVERSION (ADDRESS, MESSAGE_POINTER);
```

What does MAKE_POINTER do? In essence, it allows a value of type ADDRESS (a pre-defined type in package SYSTEM) to be stored in an object of type MESSAGE_POINTER. Since the underlying representation of MESSAGE_POINTER (for most compilers) is an ADDRESS, the function should be acceptable to the compiler. An interesting consequence of using an instantiation of UNCHECKED_CONVERSION is that a good optimizing compiler will (usually) generate no instructions for the statement

```
POINTER : constant MESSAGE_POINTER := MAKE_POINTER(something'ADDRESS);
```

but will evaluate this statement at link time and reduce it to a pre-defined literal value. POINTER, in this statement, will be a literal which contains the address of "something"; POINTER.all will generate one

instruction to retrieve the value of "something" (unless "something" is too big to get in one statement).

Therefore, to use MAKE_POINTER, we define a set of MESSAGEs, such as

```
MESSAGE_0 : constant MESSAGE(2) := (2,"OK");
```

```
MESSAGE_1 : constant MESSAGE(14) := (14, "SYSTEM RESTART"), ...
```

and define the pointer array as

```
MESSAGE_DATABASE: constant MESSAGE_ARRAY :=  
  ( 0 => MAKE_POINTER(MESSAGE_0'ADDRESS),  
    1 => MAKE_POINTER(MESSAGE_1'ADDRESS), ...
```

for all of the messages. The problem with the "new" allocator is solved, and the implementation meets its requirements (using one instruction to return the MESSAGE, with one data word per MESSAGE of overhead).

Some people may object to this last statement. After all, the function returns a MESSAGE, not a MESSAGE_POINTER. However, since MESSAGE is a composite object, the most efficient way to pass the value is by reference (and most compilers use this technique). Passing "by reference" is nothing more than passing the address of the object. Therefore, this type of approach can result in exceptionally small and fast implementations.

As to the maintainability of this approach: consider the changes needed to implement a new MESSAGE with static pointers vs. dynamic pointers. It should be readily apparent that the difference is not overly burdensome.

There are some problems with this "optimal" approach, however:

- 1) The solution requires the use of UNCHECKED_CONVERSION to convert an ADDRESS to an access value. When the 'ADDRESS attribute is used, all information relating to the base type of the object being "pointed to" is lost. The UNCHECKED_CONVERSION makes no attempt to check for compatibility between the object base type and the access type. This reduces the reliability of the software.
- 2) Certain objects take on different underlying structures depending upon how they are declared, which can cause problems when using the implementation approach. For example, an object declared with a subtype STRING(1..10) will typically be represented as 10 storage elements (bytes or words). However, an object of STRING(1..N), where N is not known until run-time, will have an extra value (a descriptor) defining the length or bounds of the string. Note that both objects are of type STRING; therefore, an object of type "access STRING" could point to either of them. The compiler cannot deduce from the access object which is to be used, thus, it must choose one (which will be wrong in half of the cases). There is a workaround for this problem; by enclosing the STRING in a discriminated record (see MESSAGE in the example), the descriptor is explicitly defined as the discriminant. This is unlikely to work for all compilers, however.
- 3) An optimization, commonly performed by compilers, is to use a register copy of a memory object for a sequence of code that does not update the memory object. Since the compiler does not know if the code is using a static pointer to update the object indirectly, the optimization technique may lead to an incorrect implementation. Experienced programmers will design their algorithms to avoid this problem; however, it can still occur.

- 4) Another problem with the solution above is that it could cause the generation of erroneous code. Specifically, if an access object is referencing a constant object, the compiler is unable to determine that the referenced object is a constant. Therefore, it might be possible to update a constant object by using the access object.

POSSIBLE SOLUTIONS:

One possible solution is to make the following changes to the LRM, which would create a function to generate access values for statically-declared objects:

- 1) Delete the sentence in LRM 3.8:8 currently stated as "An access value can only designate an object created by an allocator; in particular, it cannot designate an object declared by an object declaration."

- 2) Make the following changes within section 13.10.

- a. Renumber paragraph 13.10:4 as 13.10:6.
b. Add the following as paragraphs 13.10:4 and 13.10:5:

"The predefined generic library subprogram `UNCHECKED_ACCESS` is used to generate access values which designate objects declared by an object declaration.

"generic

type `OBJECT_TYPE` is limited private;

type `ACCESS_TYPE` is access `OBJECT_TYPE`;

function `UNCHECKED_ACCESS` (X: `OBJECT_TYPE`) return `ACCESS_TYPE`;"

- c. Add the following text as paragraphs 13.10.3:1 through 13.10.3:4:

"An access value which designates an object declared by an object declaration can be achieved by a call of a function that is obtained by instantiation of the generic function `UNCHECKED_ACCESS`.

"The effect of an unchecked access is to generate a value which matches the underlying representation of an `ACCESS_TYPE`, and which designates the object passed to the unchecked access function (or designates a descriptor associated with the object), that is, the value can be used in any way consistent with an `ACCESS_TYPE` value returned from an allocator. An implementation may place restrictions on unchecked accesses, for example, restrictions on the declarative regions or scopes in which the actual parameter for X can exist. Such restrictions must be documented in appendix F.

"Note: It is a consequence of the visibility rules that the generic function `UNCHECKED_ACCESS` is not visible in a compilation unit unless this generic function is mentioned by a with clause that applies to the compilation unit.

"References: access type 3.8, actual parameter 6.4, apply 10.1.1, declarative region 8.1, designate 3.8 9.1, generic function 12.1, generic instantiation 12.3, object declaration 3.2.1, scope 8.2, type 3.3, with clause 10.1.1

This solution attempts to repair the weak type checking of `UNCHECKED_CONVERSION`; it additionally can allow the compiler to fix the problem of inconsistent object allocation by copying the object to the heap/stack (at the cost of processor resources).

Additional features (i.e., pragma `INTERFERENCE` applied to all referenced objects or to all potential `OBJECT_TYPES`) could be added to repair the problem with the compiler optimizations. As for the updating of constants, this could either be accepted as a potential erroneous result of using `UNCHECKED_ACCESS` or additional pragmas could be used. It is not acceptable to restrict the object or access value to non-constant (or constant) items.

DIFFICULTIES TO BE CONSIDERED:

It is essential that the proposed solution be of minimal impact to existing compilers. Furthermore, the resulting semantics should make it easy for the compiler to deduce the intent of the structure and generate an efficient implementation. If the solution is not completely "clean" (i.e., may lead to an erroneous result under extreme circumstances), such a solution is still reasonable.

MUTUALLY DEPENDENT TYPES OTHER THAN ACCESS**DATE:** March 22, 1989**NAME:** E.N. Thomas**DISCLAIMER:** The views expressed in this note are those of the author, and do not necessarily represent those of SD-Scicon PLC.**ADDRESS:** SD-Scicon PLC
Pembroke House
Pembroke Broadway
CAMBERLEY
Surrey
UK
GU15 3XD**TELEPHONE:** +44 276 686200**ANSI/MIL-STD-1815A REFERENCE:** 3.8.1**PROBLEM:**

Section 3.8.1 only admits the possibility of mutually dependent types involving access types. Consequently, the rules concerning the use of incomplete types only permit use in an access type declaration (3.8.1(4)).

On the other hand, section 7.4.1(3) allows the wider use of private types before the full declaration, which permits declaration of components and parameters of the type amongst other things.

A case where mutually dependent types arises without involving access, and where the order of declaration cannot be changed to avoid the circularity concerns types in a package made visible by generic instantiation. A type needed as an actual parameter for the instantiation cannot contain a component of a type made visible by the instantiation. (It cannot be assumed that this visible type is the same for all instances.)

As an explicit example, take the predefined `DIRECT_IO`. This has a parameter `ELEMENT_TYPE`, and makes visible `COUNT`. If it is wished to have files of a record type that contain "pointers" of type `COUNT`, this cannot be declared using the following code.

```
type LIST;  
  
package LIST_IO is new DIRECT_IO (ELEMENT_TYPE => LIST);  
  
type LIST is  
record  
    VALUE : INTEGER;  
    LEFT  : LIST_IO . COUNT;  
    RIGHT : LIST_IO . COUNT;  
end record -- LIST  
;
```


IMPORTANCE: ESSENTIAL

This is no way of overcoming this problem, without using assumptions about the type about to be made visible, and probably unchecked conversions.

CURRENT WORKAROUNDS: NONE**POSSIBLE SOLUTIONS:**

Allow for the limitations of order of declaration introduced by generic instantiations, and permit wider use of incomplete types, possible in a similar way to private types prior to their full declaration.

For additional references to Section 3. of ANSI/MIL-STD-1815A, see the following sections, revision request numbers, and revision request titles in this document.

REVISION REQUEST

<u>NUMBER</u>	<u>TITLE</u>	<u>SECTION</u>
0093	CONSTANTS REFERRED TO PACKAGE BODY	7
0096	LIMITATIONS ON USE OF RENAMING	8
0012	MUTATION OF TYPES	9
0101	IMPLEMENTATION OF EXCEPTIONS AS TYPES	11

**ADA 9X REVISION REQUESTS
THAT REFERENCE
ANSI/MIL-STD-1815A**

SECTION 4. NAMES AND EXPRESSIONS

AGGREGATE FOR NULL RECORDS AND NULL ARRAYS**DATE:** March 17, 1989**NAME:** Mats Weber**ADDRESS:** Swiss Federal Institute of Technology
EPFL DI LITH
1015 Lausanne
Switzerland**TELEPHONE:** 0041 21 693 11 11
E-mail: madmats@elma.epfl.ch**ANSI/MIL-STD-1815A REFERENCE:** 4.**PROBLEM:**

There is no way of writing aggregates for null records and null arrays.

Example:

```
type Void is -- such a type is often useful as a generic record -- actual parameter when
some of the functionality null; -- of the generic unit is not required.
end record;
```

```
function Void_Value return Void is
```

```
begin
return ...; -- how should I write this ?
end Void_Value;
```

For array types, the problem is a little bit different: An aggregate for a null array can be written if some value of the component type is available (as in `String'(1..0 => 'a')`), but this is not true for example if the component type is a generic formal private type.

IMPORTANCE: IMPORTANT**CURRENT WORKAROUNDS:**

Declare a variable in a local declare block and use its value.

```
function Void_Value return Void is
```

```
Result : Void; -- cannot be a constant because constants -- must be initialized.
begin
return Result;
end Void_Value;
```

POSSIBLE SOLUTIONS:

Syntax for a null array aggregate: (<Lower_Bound>..**<Upper_Bound>**) without component association.
Would raise Constraint_Error if **<Upper_Bound>** >= **<Lower_Bound>**.

Syntax for a null record aggregate: (null).

Note that this can cause some problems with overload resolution.

SEPARATION OF EXPONENT AND MANTISSA**DATE:** December 9, 1988**NAME:** Lawrence J. Gallaher**ADDRESS:** LMSC 5T 40
1150 Gemini Ave.
Houston, TX 77058**TELEPHONE:** (713) 282-6305**ANSI/MIL-STD-1815A REFERENCE:** 4., 13.**PROBLEM:**

One of the problems of writing in Ada a machine independent elementary function library (sin, cos, sqrt, etc..) is the mapping or factoring of a real number into a reduced range, in particular, separating a real number into its exponent and mantissa. For example, for a given real X (not zero), one would like to find the X_0 and J such the $X = X_0 \cdot 2^J$, where J is integer and $0.5 \leq X_0 < 1.0$. This is straight forward in assembler code by picking off the appropriate bits from the real representation of X . However since each machine hardware has a different representation for its reals (and even different kinds of reals, such as 32, 64, or even 128 bit reals) there is no easy or efficient way to do this in Ada. One needs to do this separation of exponent and mantissa for quick evaluation of the square root, cube root, and the logarithm functions, and one would like to do it efficiently in a machine independent way.

IMPORTANCE: ADMINISTRATIVE

This is an efficiency problem. If it is not solved then elementary libraries containing sqrt, ln, etc.. will be either quite inefficient, or highly machine dependent-- this is the situation now.

CURRENT WORKAROUNDS:

At present a procedure Decompose(X, J) can be written which does the job but is very slow for large values of X :

```
procedure Decompose(X : in out real; J : in out integer) is --
--This procedure takes in an X and returns an X and J
--such that X[in] = X[out]*2**J. X[out] in 0.5..1.0 if X /=0.0.
begin
    J:=0;
    if X /= 0.0 then
        while abs(X) >= 0.0 loop X := X/2.0; J := J + 1; end loop;
        while abs(X) < 0.5 loop X := X*2.0; J := J - 1; end loop;
    end if;
end Decompose;
```

This is machine independent and works for all X . This is a linear search for J ; going to a binary search for J will speed things but that's still gross. The same thing can be accomplished in just a few assembler language instructions with no need even for any floating point arithmetic operations.

POSSIBLE SOLUTIONS:

Make the procedure `Decompose(X, J)` or something equivalent an intrinsic of the Ada language. It would be very easy for the compiler to generate the appropriate machine code to do this for a particular hardware and do it efficiently-- if it is a part of the language and done through the compiler this would effectively achieve machine independence.

It might also be desirable to generalize this process to be able to decompose on a radix other than 2.

This suggested modification to Ada would be upward compatible with the existing standard.

ATTRIBUTE P'REPRESENTATION**DATE:** April 4, 1989**NAME:** Eric F. Heck**ADDRESS:** GE Aerospace GCSD
Mail Stop 13-7-5
Camden, NJ 08102**TELEPHONE:** (609) 338-4776**ANSI/MIL-STD-1815A REFERENCE:** 4.1.4(4)**PROBLEM:**

An attribute P'Representation is needed for enumeration types. This attribute would return the underlying value of an object P. The result is of Universal_Integer. This attribute would return the representation value for an object. For example:

```
type Network_Type is
    (UHF_High, IWC, UHF_Low, VHF_High, VHF_Low);
```

```
for Network_Type use (UHF_High => 16#02#,
                      IWC       => 16#04#,
                      UHF_Low   => 16#10#,
                      VHF_High  => 16#12#,
                      VHF_Low   => 16#14#);
```

```
Value : Integer;
```

```
Network : Network_Type := IWC;
```

```
  :
  :
```

```
Value := Network'Representation; -- Value now equals 16#04#
```

IMPORTANCE: ESSENTIAL

The Ada LRM allows placing representation clauses on types, however, there is no way to get that assigned value back. Processing may need to be performed on objects which requires knowing its assigned underlying representation. If this request is not satisfied, certain applications may not choose to use Ada, or if they do use Ada it would be poor Fortran looking Ada.

CURRENT WORKAROUNDS:

We don't use enumeration types for objects whose underlying representation may need to be known. We declare those objects as integer or byte type.

ALLOW OTHERS WITH NAMED ASSOCIATION AT ARRAY INITIALIZATION**DATE:** January 15, 1989**NAME:** William Thomas Wolfe**ADDRESS:** Home: 102 Edgewood Avenue #2
Clemson, SC 29631 USAOffice: Department of Computer Science
Clemson University
Clemson, SC 29634 USA**TELEPHONE:** Home: (803) 654-7030
Office: (803) 656-2847
E-mail: wtwolfe@hubcap.clemson.edu**ANSI/MIL-STD-1815A REFERENCE:** 4.3.2(6)**PROBLEM:**

Cannot use an others choice in conjunction with named associations during the initialization of an array at the point of declaration.

CONSEQUENCES:

The programmer must, for example, enumerate 32 TRUEs in order to get to a point at which the 33rd element of a Boolean array can be set to FALSE and an others choice can finally be used to set the remaining elements of the array to TRUE.

CURRENT WORKAROUNDS: NONE**POSSIBLE SOLUTIONS:**

Revise ANSI/MIL-STD-1815A 4.3.2 to permit the use of an others choice in conjunction with named association in all situations in which the precise meaning of "others" is obvious from context.

ALLOW RELATION TO SPECIFY NONCONTINUOUS RANGE

DATE: January 15, 1989

NAME: William Thomas Wolfe

ADDRESS: Home: 102 Edgewood Avenue #2
Clemson, SC 29631 USA

Office: Department of Computer Science
Clemson University
Clemson, SC 29634 USA

TELEPHONE: Home: (803) 654-7030
Office: (803) 656-2847
E-mail: wtwolfe@hubcap.clemson.edu

ANSI/MIL-STD-1815A REFERENCE: 4.4(2)

PROBLEM:

It is not possible to specify "if X in 3 | 4 | 6 | 9 then..."; only a continuous range can be specified.

CONSEQUENCES:

The workaround described below must be used, reducing clarity.

CURRENT WORKAROUNDS:

An extended Boolean expression such as

if (X = 3) or (X = 4) or (X = 6) or (X = 9) then...

can be used, but this is unnecessarily verbose.

POSSIBLE SOLUTIONS:

Add a new production to the definition of "relation":

```
relation ::= simple_name [not] in
           simple_expression { | simple_expression }
```

SHIFT AND ROTATE OPERATIONS FOR BOOLEAN ARRAYS**DATE:** June 27, 1989**NAME:** Neil Salant**ADDRESS:** ITT Avionics
Dept. 73813
390 Washington Ave.
Nutley, NJ 07110**TELEPHONE:** (201) 284-3896**ANSI/MIL-STD-1815A REFERENCE:** 4.5.1**PROBLEM:**

Ada does not directly provide shift and rotate operations for boolean arrays. Packed boolean arrays are often used to represent hardware types such as "byte" and "word". Ada provides logical functions such as AND, OR, NOT, and XOR for such objects but does not make provisions for logical SHIFT and ROTATE operations. These operations are commonly used in a number of applications including signal processing, communications, and cryptography.

IMPORTANCE: IMPORTANT

The lack of these operations will result in real-time Ada applications which require shift and rotate operations turning to other less desirable means of getting this feature. Usually these means are not as efficient, and introduce a number of problems. A uniform Ada capacity to perform these functions is preferable to such options for reasons of portability, efficiency, consistency, and reliability. These operations are often performed in a time critical environment, where inefficiency may have serious impact on the overall performance of the application.

CURRENT WORKAROUNDS:

1. **Assembly Language Interface:** Call an assembly language routine to perform a single assembly language instruction. This involves subroutine call overhead which can be avoided by an inline assembly language insertion. Either of these methods is implementation dependent, and thus non portable. Problems introduced by assembly language interface include loss of strong typing, lack of implicit constraint checking, undefined exception handling across the interface, and for inline assembler, implementation dependence on determining the address of the operand.
2. **Ada Workaround:** Shift and rotate operations can be performed in Ada, but require several (two or three) Ada assignment statements for boolean array slices. Unfortunately, many (all?) compilers do not recognize the intent of this series of assignments, and generate a series of set/clear bit instructions. Thus for an typical packed boolean array of 16 (or 32) elements, or more instructions to simulate an operation which often requires only a single assembly language instruction. This is clearly inefficient, and may introduce timing problems for low latency processing applications.

POSSIBLE SOLUTIONS:

Introduce SHIFT and ROTATE operations for boolean array operands of arbitrary size. Such operations on packed boolean arrays of sizes equal to basic hardware types (ie. byte, word, longword) should be guaranteed to be implemented in a single machine instruction, assuming that the instruction exists in the target instruction set.

REMAINDER DIVIDE FOR REAL NUMBERS**DATE:** May 5, 1989**NAME:** Lawrence J. Gallaher**ADDRESS:** LMSC 5T 40
1150 Gemini Ave.
Houston, TX 77058**TELEPHONE:** (713) 282-6305**ANSI/MIL-STD-1815A REFERENCE:** 4.5.5**PROBLEM:**

One of the problems that comes up in writing in Ada a machine independent elementary function library (sin, cos, etc.) is the mapping of wraparound of a real number model another real number (e.g., $x \bmod (2\pi)$, needed for the trig functions). A remainder divide for reals is not now available in Ada and is needed.

IMPORTANCE: ADMINISTRATIVE

This is an efficiency problem. If not solved then elementary function libraries containing sin, cos, etc. will be slow, inaccurate, and/or machine dependent -- this is the situation now -- see current workarounds below.

CURRENT WORKAROUNDS:

At present the remainder divide for reals can be done by the rather crude statement

$$x := x - \text{Integer}(x/(2\pi));$$

There are however three things wrong with this method:

1. It is slow, especially on hardware that has a built in remainder divide for reals.
2. It can lead to significant rounding errors in the result. See the Sym, Wichmann, Kok, Winters chapter in "Scientific Ada," by Ford, Kok, and Rogers, Cambridge Univ Press, 1986, for a more accurate but much more time consuming solution.
3. It can lead to `numeric_error` or `constraint_error` exception if $\text{Integer}(x/(2\pi))$ happens to exceed the largest integer in the machine, a not uncommon occurrent. No constraint error should ever be raised here as long as the type of x spans at least $-\pi..pi$.

POSSIBLE SOLUTIONS:

Make the operator "rem" (or "mod") applicable to real numbers in Ada, and let the compiler produce the 'best-for-that-machine' code for the remainder divide of reals. The result of $x \text{ rem } b$ ought to be in the interval $-|b|/2..|b|/2$.

This suggested modification to Ada would be upward compatible with the existing standard.

EXPONENTS OF ZERO BY A ZERO EXPONENT

DATE: October 11, 1988

NAME: David M. Battany

ADDRESS: 7604 Concord Lane NE - #B203
Bremerton, WA 98310

TELEPHONE:

ANSI/MIL-STD-1815A REFERENCE: 4.5.6(6)

PROBLEM:

The reference manual indicates that exponentiation of zero by a zero exponent delivers the value one; however, $0^{**}0$ is an indeterminate form.

IMPORTANCE:

CURRENT WORKAROUNDS:

POSSIBLE SOLUTIONS:

Consider making $0^{**}0$ (likewise $0.0^{**}0$) raise an exception.

VISIBILITY OF HIDDEN IDENTIFIERS IN QUALIFIED EXPRESSIONS**DATE:** May 23, 1989**NAME:** Jeffrey R. Carter**ADDRESS:** Martin Marietta Astronautics Group
MS L0330
P.O. Box 179
Denver, CO 80201**TELEPHONE:** (303) 971-4850
(303) 971-6817**ANSI/MIL-STD-1815A REFERENCE:** 4.7(3)**PROBLEM:**

A qualified expression does not provide visibility to hidden identifiers.

IMPORTANCE: ADMINISTRATIVE**CURRENT WORKAROUNDS:**

Use the hidden identifier's full name.

POSSIBLE SOLUTIONS:

Allow the evaluation of a qualified expression to search enclosing scopes for hidden identifiers which have the specified type.

```
procedure alpha is
  type bravo is (charlie);
  type delter is array (bravo) of natural;
  procedure echo (foxtrot : in delter) is
    charlie : natural;
  begin -- echo
    charlie := foxtrot (bravo'(charlie) );
  end echo;

begin -- alpha
  null;
end alpha;
```

The qualified expression [bravo'(charlie)] clearly indicates the desire to refer to standard.alpha.charlie, which is hidden by standard.alpha.echo.charlie. This should be allowed. It does not detract from readability, and should not be difficult to implement. Such a change would be compatible with the current standard.

GARBAGE COLLECTION

DATE: June 7, 1989

NAME: Neal Altman, (ACM Special Interest Group on Ada, Ada Runtime Environment Working Group)

ADDRESS: Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

TELEPHONE NUMBER: (+12) 268-7613
E-mail: (ARPAnet) na@sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 4.8

PROBLEM:

Garbage collection, when present, may require unbounded execution time. An Ada application has limited control over the initiation of garbage collection, and none over its duration. Ada semantics should support schemes for controlled reclamation of deallocated storage.

SPECIFIC REQUIREMENT/SOLUTION CRITERIA:

Long duration applications cannot tolerate loss of storage, either by failure to return storage to the available storage pool or due to fragmentation of free storage into unusable small units. High performance applications require that storage be freed within predictable time limits. A time critical application cannot be preempted for an unbounded length of time by a garbage collector.

The Ada Reference Manual (ARM) should provide explicit user control of the garbage collection process, so that a user may:

- 1) Control how garbage collection starts: never, under user control or automatically.
- 2) Tailor the automatic execution of garbage collection (e.g. the starting time or triggering event, priority, cycle, maximum duration).
- 3) Exert dynamic control of garbage collection (e.g. vary the mode of garbage collection, depending upon application requirements, during application execution).

The ARM should also constrain the definition of the process of garbage collection so that, where present, garbage collection is defined to return all deallocated storage to the free pool. It would also be useful to define when garbage collection will be completed under given conditions of execution.

The ARM must also define explicit operations for the user to determine the status of the storage pool, so the user may initiate garbage collection when required.

IMPORTANCE:

Applications with hard deadlines will continue to avoid garbage collection services provided by Ada implementations. Individual applications will implement deallocation using either non-standard facilities or with the `UNCHECKED_DEALLOCATION` procedure.

CURRENT WORKAROUNDS:

The Ada language provides for the dynamic creation of objects by the use of access types and allocators. After allocation, these objects remain in existence for as long as the object can be accessed. After all references to the object are removed, the ARM specifies that "An implementation may (but need not) reclaim the storage occupied by an object created by an allocator, once this object has become inaccessible" (ARM 4.8.7, p. 4-5).

If garbage collection is supported by an implementation, the application may control the initiation of deallocation of dynamically created objects by:

- 1) deferring the clearing of access values for objects;
- 2) using pragma `CONTROLLED` (if supported by the implementation);
- 3) performing deallocation explicitly with the `UNCHECKED_DEALLOCATION` procedure.

In addition to objects explicitly created by applications, the Ada runtime may create additional objects implicitly, by entering the scope of a declaration, during procedure calls and so on. Implementors can use a variety of schemes for the allocation and deallocation of such objects, including the use of a common storage pool and garbage collection. The user presently can control the initiation of deallocation for such objects by not leaving the scope of the program unit causing their creation.

Once garbage collection begins, the application has no control over the duration of the deallocation process. Since the effective priority of garbage collection is unspecified, deallocation may allow the suspension of application execution. This characteristic is unacceptable for certain categories of applications which might reasonably be expected to make use of dynamically allocated objects.

The ARM also provides no assurance that garbage collection (if present) will return all allocated storage for reuse.

Applications which perform time critical processing, particularly those with hard deadlines, need to control interruptions of critical processing. These applications can often make effective use of dynamic storage.

In radar systems, an application might dynamically create and delete data records for tracked aircraft. Such records must be rapidly updated and displayed. The imposition of large delays for storage reclamation can result in critical delays in data processing and in turn, affect the safety of aircraft and installations dependent upon radar data.

Long duration applications must manage their storage pool so that the pool of storage can always be re-used. Fragmentation must be controlled so that new objects of useful size can be allocated. All deallocated storage must be returned to the pool in a timely fashion, to prevent eventual exhaustion of free space.

Typically, in safety critical and hard deadline systems, storage is statically allocated to an application defined storage pool, and appropriate routines are written to allocate and deallocate from this static pool. This makes the allocation of data predictable and safe, but loses the generality and flexibility of a common storage pool.

POSSIBLE SOLUTIONS:

Explicit control of garbage collection might be provided by means of a package interface. The following example assumes that garbage collection is performed by an Ada task. The garbage collection task is initiated and called by the runtime, unless the user exerts specific control.

```
with SYSTEM;

package GARBAGE_CONTROL is
  type COLLECTION_MODES is (DISABLED, USER_CONTROL, AUTOMATIC);

  type POOL_UNITS is range 0..SYSTEM.MEMORY_SIZE;

  function FREE_POOL return POOL_UNITS;

  function BIGGEST_FREE_CHUNK return POOL_UNITS;

  task GARBAGE_COLLECTOR is
    entry INITIALIZE(
      MODE           : in COLLECTION_MODES;
      PRIORITY       : in SYSTEM.PRIORITY;
      MAXIMUM_DURATION : in DURATION);
    entry COLLECT;
    entry COLLECT(
      COLLECTION_DURATION : in DURATION);
  end GARBAGE_COLLECTOR;

end GARBAGE_CONTROL;
```

The INITIALIZE entry of the GARBAGE_COLLECTOR task may only be called once. It sets the operation of the collector to one of three defined modes:

DISABLED - Garbage collection is deactivated. Rendezvous with the collection task will fail.

USER_CONTROL - The user exerts sole control over the activation of the garbage collector

AUTOMATIC - System may activate the garbage collector within the constraints provided by the other arguments. The user may still activate the garbage collector.

Two constraints are provided. PRIORITY sets the priority of the garbage collection task. MAXIMUM_DURATION limits the maximum time the garbage collector may consume in performing a single garbage collection run. This value may be overridden during any activation of the collector. The duration is only effective after the start of the collection task, but includes any time consumed by preemption of the collector by higher priority tasks.

The COLLECT entry point of the task starts garbage collection. The duration of execution will not exceed COLLECTION_DURATION, if provided, otherwise the duration of execution will not exceed MAXIMUM_DURATION.

FREE_POOL returns the number of storage units available for allocation in the storage pool.

`BIGGEST_FREE_CHUNK` returns the number of storage units available in the largest contiguous free block of storage in the storage pool.

Note that this package is written as if dynamic priorities were available. Using standard Ada, some other mechanism would be required to ensure user control over the priority of the garbage collection task. Use of a generic package is precluded by the need to prevent multiple garbage collector tasks.

DIFFICULTIES TO BE CONSIDERED:

Effective garbage collection often depends upon the compaction of storage as well as the deallocation of individual chunks of storage. Many useful compaction schemes are most effective as atomic operations, and are difficult or impossible to implement as interruptible operations.

The extent to which a given application can tolerate temporary preemption for garbage collection varies, as does the amount of processing time available for any collection scheme. It is often feasible to trade increased processing time for limited preemption of application code, but the optimal trade off point varies from application to application.

Some actions performed during garbage collection are typically nonpreemptible (e.g. during compaction, moving a variable and updating all access variables which point to it). The duration of such nonpreemptible segments is a critical characteristic of any garbage collection scheme, but is determined by the collection algorithm. If garbage collection has an effective priority, it should be possible for higher priority tasks to preempt its function. In this case, the application wishes to control the duration of nonpreemptible segments as well as (or even in preference to) the total amount of time consumed by the garbage collector. In any case, implementors should be encouraged to minimize the duration of any individual nonpreemptible code segment.

In many Ada environments, particularly virtual memory systems, the Ada runtime may dynamically modify pool size, in accordance with application needs. Thus the free storage pool may increase (or decrease) in size. Such schemes, particularly those with demand paging, may simplify pool management, but complicate performance and control of garbage collection. Applications now must trade the overhead of garbage collection against allowing pool growth into a larger, heterogeneous memory space.

REFERENCES:

- [1] N. H. Weideman, et. al., Ada for Embedded Systems: Issues and Questions; Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh PA 15213, CMU/SEI-87-TR-26, ESD-TR-87-189; December 1987.

TYPE CONVERSION OF STATIC TYPE CAN BE NON-STATIC**DATE:** March 22, 1989**NAME:** E.N. Thomas**DISCLAIMER:** The views expressed in this note are those of the author, and do not necessarily represent those of SD-Scicon PLC.**ADDRESS:** SD-Scicon PLC
Pembroke House
Pembroke Broadway
CAMBERLEY
Surrey
UK
GU15 3XD**TELEPHONE:** +44 276 686200**ANSI/MIL-STD-1815A REFERENCE:** 4.9**PROBLEM:**

An implicit type conversion does not make a static expression non-static, but an explicit conversion does. Consequently, if an attribute like 'LAST of a original type is used to supply the bounds to declare a subtype of another numeric type, then the conversion of the attribute value makes the value non-static, even the original type was static.

This leads to problems in defining certain forms of types that are related to original types, because the resulting types can become non-static unless the original types were defined in a specific style (and if the original type is in a predefined package, then that style cannot be obtained). Such a relationship may be needed to define composite structures whose shapes or sizes are functions of the bounds of the (several) original type(s) -for example a character position within a file has bounds dependent on the ranges of columns, lines, and pages allowed in the file. If the usage of the structure is such as to require a static type, then expressing the relationship between the types becomes difficult or impossible.

This is illustrated by the attached example. The aggregates used in PNS involve illegal use of others, because subtype PIndex is not static, although the bounds of Index from which it is defined are static.

package Arrays is

Max : constant := 12;

type Index is range 0 .. Max;

subtype Part is Index range 1 .. Max;

type PSpec is array (Part) of BOOLEAN;

subtype PIndex0 is NATURAL

range NATURAL(Index'FIRST) .. NATURAL(Index'LAST)

subtype PIndex is PIndex0

range PIndex0(Part'FIRST) .. PIndex0'LAST;

```

type PList is array (NATURAL range <>) of PSpec;
subtype PSet is PList (PIndex);

```

```

Place : constant BOOLEAN := FALSE;
Cross : constant BOOLEAN := TRUE;

```

```

EmptyP : constant PSpec := (others => FALSE);

```

```

PNS : PSet := PSet'(1 => PSpec'(1|2 => Place,
                               3|4 => Cross,
                               others => Place),
                  others => EmptyP
);

```

```

end Arrays;

```

IMPORTANCE: ESSENTIAL

Regularity is compromised without these, and the design goals of 1.3(3) are violated ("a small number of underlying concepts integrated in a consistent way").

CURRENT WORKAROUNDS:

To avoid explicit conversions in such examples, it is necessary to be able to use universal values for the relevant bounds. This in turn requires that the original types are declared with accompanying named numbers for this use, as in the following example.

```

Max : constant := 12;

IndexFIRST : constant := 0;
IndexLAST  : constant := Max;
PartFIRST  : constant := 1;
type Index is range IndexFIRST .. IndexLAST;
subtype Part is Index range PartFIRST .. IndexLAST;

type PSpec is array (Part) of BOOLEAN;
subtype PIndex0 is NATURAL
    range IndexFIRST .. IndexLAST;
subtype PIndex is PIndex0
    range PartFIRST .. PIndex0'LAST;
type PList is array (NATURAL range <>) of PSpec;
subtype PSet is PList(PIndex);

```

The big problem with this is that the original type needs to be declared with advance knowledge of this requirement. When the original declaration did not follow this style, there can be severe problems, particularly when it cannot be changed because it is in a package that is "inviolable" for the current use - which is always the case for predefined packages such as TEXT_IO.

In some cases the types and subtypes can be re-arranged to use derived types, but this is not always satisfactory.

POSSIBLE SOLUTIONS:

The definition of static expressions should be such that static properties are not lost by operations that clearly do not disturb these properties. So type conversion from a static (sub)type to another static (sub)type should not make an expression non-static.

Further, whether or not an attribute is static should depend on whether or not the relevant value is static, and not on the sort of attribute. For example, 4.9(13) says array attributes are not static-but if the relevant bound of the actual array or type is static, then the corresponding attribute should be, and vice versa.

SOME CONVERSIONS SHOULD BE STATIC

DATE: October 28, 1988

NAME: S. Tucker Taft

ADDRESS: Intermetrics, Inc.
733 Concord Avenue
Cambridge, MA 02138

TELEPHONE: 617-661-1840
E-mail: stt@inmet.inmet.com
E-mail: uunet!inmet!stt

ANSI/MIL-STD-1815A REFERENCE: 4.9(2)

PROBLEM:

Conversion should be considered static when the value is discrete and static, and the target subtype is discrete and Static. there are several places in the language where static expressions are required, and there are many times when it would be advantageous to be able to use type conversion in these places.

IMPORTANCE:

It will be less efficient and/or more complex to provide static constants for derived types based on similar constants available for the parent type.

CURRENT WORKAROUNDS:

The current restriction making discrete conversion non-static seems arbitrary and non-intuitive.

Static discrete conversion can be accomplished currently by using the circuitous route of a qualified expression above a "POS" attribute call for integer types, and a 'VAL above a 'POS attribute call for enumeration types. This workaround is particularly ugly, and there seems no reason why the language should not consider the direct conversion static.

POSSIBLE SOLUTIONS:

Explicitly include conversion to a static discrete subtype of a static discrete expression within the list of primaries allowed by 4.9 (2) for static expressions.

DEFINITION OF STATIC SUBTYPE**DATE:** March 8, 1989**NAME:** William Thomas Wolfe

ADDRESS: Home: 102 Edgewood Avenue #2
Clemson, SC 29631 USA
Office: Department of Computer Science
Clemson University
Clemson, SC 29634 USA

TELEPHONE: Home: (803) 654-7030
Office: (803) 656-2847
E-mail: wtwolfe@hubcap.clemson.edu

ANSI/MIL-STD-1815A REFERENCE: 4.9(11)**PROBLEM:**

The cited reference reads: "A static subtype is either a scalar base type, other than a generic formal type...". This means that even though a value can be completely determined at compile time, such a value is not necessarily considered a static expression.

CONSEQUENCES:

In programming situations in which it is necessary to define a data type whose range depends upon the characteristics of a generic formal parameter, Ada fails to provide the necessary support.

Example:

```
1      with GENERIC_LINKED_LIST, SYSTEM;
2
3
4      generic
5
6
7      type DATA_ITEM is limited private;

(some lines which are not relevant to the example deleted)

23     package GENERIC_ADAPTIVE_HASHING_PACKAGE is
24
25
26     type ADAPTIVE_HASHING_KEY is range 0..SYSTEM.MAX_INT/2;
    (some comment lines deleted)

32     type ADAPTIVE_HASHING_STRUCTURE is limited private;
33
34
35     type DATA_RECORD is
```

```

36  record
37  ACCESS_KEY_VALUE : ADAPTIVE_HASHING_KEY;
38  DATA_ITEM_VALUE : DATA_IT
39  end record;

      (rest of the package spec follows, including procedures and functions having parameters of
      type DATA_RECORD)

1    with DISK_DRIVE_SIMULATOR, SYSTEM, UNCHECKED_CONVERSION;
2
3
4    package body GENERIC_ADAPTIVE_HASHING_PACKAGE is
5
6
7    use DATA_RECORD_HANDLER;
8
9
10   type CYLINDER_NUMBER_RANGE is range 0..63;
11
12   type TRACK_NUMBER_RANGE    is range 0..7;
13
14   type SECTOR_NUMBER_RANGE   is range 0..3;
15
16   type STORAGE_UNIT_RANGE    is range 0..255;
17
18   package DISK_DRIVER
19   is new DISK_DRIVE_SIMULATOR (CYLINDER_NUMBER_RANGE,
      TRACK_NUMBER_RANGE, SECTOR_NUMBER_RANGE, STORAGE_UNIT_RANGE);

20
21   use DISK_DRIVER;
22
23
24   NUMBER_OF_DATA_RECORDS_PER_DATA_PAGE : constant POSITIVE :=
      (DISK_DRIVER.DATA_PAGE'SIZE
25   - DISK_DRIVER.DATA_PAGE_ADDRESS'SIZE - 8) / DATA_RECORD'SIZE;
26   (some comment lines deleted from the example)
27   type DATA_RECORD_STORAGE_RANGE is range
28   1..NUMBER_OF_DATA_RECORDS_PER_DATA_PAGE;
      -----
29   >>> Static value required <4.9>.

```

Although it is possible to legally define type DATA_RECORD_STORAGE is array (1..NUMBER_OF_DATA_RECORDS_PER_DATA_PAGE) of DATA_RECORD; we run into trouble when we try to say type DATA_PAGE_BUFFER is record DATA_RECORDS : DATA_RECORD_STORAGE; LAST_FILLED_DATA_RECORD_SLOT : DATA_RECORD_STORAGE_RANGE; -- (other fields...) end record;

As a consequence, it is not possible to store an index into the DATA_RECORDS array, even though this is an array whose size is completely determined at compile time.

CURRENT WORKAROUNDS:

In the above example, the workaround is to either linearly scan or binary-search a companion array of BOOLEAN values which are true if the corresponding data record slot is filled. It is always the case that the full slots are contiguous and left-justified in the array, so it would be more efficient to store and maintain an index into the array which tells us the number of the last slot which is filled with valid data.

POSSIBLE SOLUTIONS:

Revise ANSI/MIL-STD-1815A 4.9 (11) such that the phrase "other than a generic formal type" is deleted, leaving the third sentence to read "A static subtype is either a scalar base type, or a scalar subtype formed by imposing on a static subtype either a static range constraint or a floating or fixed point constraint whose range constraint, if any, is static."

For additional references to Section 4. of ANSI/MIL-STD-1815A, see the following sections, revision request numbers, and revision request titles in this document.

REVISION REQUEST

<u>NUMBER</u>	<u>TITLE</u>	<u>SECTION</u>
0045	OVERFLOW AND TYPE CONVERSION	3
0070	USER DEFINED ASSIGNMENT	7

**ADA 9X REVISION REQUESTS
THAT REFERENCE
ANSI/MIL-STD-1815A**

SECTION 5. STATEMENTS

FAULT TOLERANCE

DATE: June 7, 1989

NAME: Larry Lehman (ACM Special Interest Group on Ada, Ada Runtime Environment Working Group)

ADDRESS: Integrated Systems Inc.
2500 Mission College Blvd.
Santa Clara, CA 95054

TELEPHONE: (408) 980-1500
E-mail: (ARPAnet) llehman@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 5., 6., 9., 11.

PROBLEM:

The pervasiveness of failure handling in mission critical applications requires appropriate support for fault tolerance and survivability for existing language operations.

SPECIFIC REQUIREMENT/SOLUTION CRITERIA:

Mission critical embedded systems, including single processor, multiprocessor, and distributed environments, must be able to withstand hardware component failures (or software component failures that may be indistinguishable from hardware failures) during execution to the extent supported by the surviving hardware. This requires sufficient language support to allow applications to implement error detection, damage confinement, assessment, error recovery and continued operation.

- 1) The language shall contain a model for partial failure of an Ada program that includes at least: task operations, operators and subprogram calls, and data object accesses.
- 2) The model shall define semantics sufficient to permit Ada applications to detect, confine, assess, report, and recover from these partial failures.

IMPORTANCE:

Support for fault tolerant and survivable applications may require significant and ad-hoc coding performed outside of the Ada language.

CURRENT WORKAROUNDS:

At least two workarounds are currently in use: multiprogramming and interrupt-based exception handling.

In the first, the developer uses separate communicating programs to compose the application. Failure of one program to communicate with another program will be detected by the interprogram communication services. Program failure recovery actions may result in reloading and restarting a program.

In the second, some of the interrupt handling routines are mapped by the runtime system into exceptions that are detected within the program by the exception handling mechanism.

Both of these mechanisms have significant limitations. An application which uses multiprogramming loses much of the safety of Ada in interprogram communication. In addition, the use of multiprogramming alone cannot detect the failures that occur within a program. The second mechanism is very hardware dependent, requires changes to the runtime system, and potentially overloads exceptions. Neither mechanism supports portability and maintainability.

POSSIBLE SOLUTIONS:

Addition of a model, with sufficient failure semantics for existing language features, to the Ada Reference Manual. This model will also allocate responsibility to support failure semantics between the compilation system and the Run-Time System.

REFERENCES/SUPPORTING MATERIAL:

- [1] J.C. Knight and J.I.A. Urquhart, "On the Implementation and Use of Ada on Fault-Tolerant Distributed Systems," IEEE Transactions on Software Engineering, Vol. SE-13, No. 5, May 1987.

REFERENCE TO VARIABLE NAMES**DATE:** March 8, 1989**NAME:** William Thomas Wolfe**ADDRESS:** Home: 102 Edgewood Avenue #2
Clemson, SC 29631 USAOffice: Department of Computer Science
Clemson University
Clemson, SC 29634 USA**TELEPHONE:** Home: (803) 654-7030
Office: (803) 656-2847
E-mail: wtwolfe@hubcap.clemson.edu**ANSI/MIL-STD-1815A REFERENCE:** 5.2, 2.1(6)**PROBLEM:**

There is no convenient method for referring to the `variable_name` appearing on the left-hand-side of an assignment statement; if the `variable_name` appears in the right-hand-side as well, which is frequently the case, the name of the variable must be retyped.

CONSEQUENCES:

Rather verbose statements are necessary:

```

LONG_AND_DESCRIPTIVE_VARIABLE_NAME
:= LONG_AND_DESCRIPTIVE_VARIABLE_NAME
+ LONG_AND_DESCRIPTIVE_VARIABLE_NAME ** (-3.1415926);

```

CURRENT WORKAROUNDS: NONE**POSSIBLE SOLUTIONS:**

Add the tilde character, `~`, to the list of special characters defined in 2.1 (6). In 5.2, define the meaning of the `~` in the context of an assignment statement's right-hand-side as the current contents of the `variable_name` which appears on the left-hand-side of the assignment statement. This simplifies assignment statements considerably:

```
LONG_AND_DESCRIPTIVE_VARIABLE_NAME := ~ + 1;
```

```
LONG_AND_DESCRIPTIVE_VARIABLE_NAME := ~ + ~ ** (3.1415926);
```

This in turn makes it more probable that long and descriptive variable names will be used by programmers, leading to a general improvement in the level of documentation in Ada-based software systems.

The reason for the selection of the tilde character in particular is that we wish to follow the convention already used in dictionaries of the English language. Dictionary usage takes the form:

natural (adj.) [...definitions of natural...]
-- ~ly (adv.) [...definition of naturally...]
-- ~ism (n.) [...definition of naturalism...]

We thereby make the Ada usage seem easy to learn, as a consequence of the learner's ability to draw an analogy to the already-established English dictionary usage of roughly the same concept.

USER DEFINED ASSIGNMENT STATEMENT FOR LIMITED PRIVATE TYPES

DATE: April 28, 1989

NAME: Christoph Grein
Member of Ada Germany

ADDRESS: Hauptstr. 42
D-8919 Greifenberg
FRG

TELEPHONE: 0049/8192/1411

ANSI/MIL-STD-1815A REFERENCE: 5.2(3), 5.2.1(2)

PROBLEM:

Ada UK/005 Language Change Request to introduce user defined assignment statement for limited private types

With redefinition of assignment for limited private types allowed, the following two requirements of the language standard can no longer be guaranteed:

5.2(3) The expression is evaluated before the assignment.

5.2.1(2) In case that CONSTRAINT_ERROR is raised, the value of the variable is left unchanged.

As an example, consider ADT (a limited private type from the user's point of view) being defined locally as

```
type ADT is
  record
    A, B: BOOLEAN;
  end record;
```

```
A: ADT := (A => FALSE, B => TRUE);
```

Now, let the new syntax allow for

```
procedure ":=" (LEFT: [in] out ADT; RIGHT: in ADT) is
begin
  LEFT.A := RIGHT.B;
  LEFT.B := RIGHT.A;
end ":=";
```

The assignment statement `A := A;` produces, assuming a "call by reference" parameter passing mechanism, a result which does not comply with the requirements of the standard. (Cf. the example on "tar sauce" in 5.2.1 (3..4).)

Although the example given above may look rather contrived, the problems exists. Either the above mentioned requirements have to be loosened or there can be no such language standard requirements apply.

(Alternatively, one might think of requesting that for procedures destined to serve as assignment statement the parameter passing mechanism be "copy in" for RIGHT and "[copy in] copy out" for LEFT. There might be further problems implied with this (e.g. when does the compiler know about this - consider renaming).)

IMPORTANCE:**CURRENT WORKAROUNDS:****POSSIBLE SOLUTIONS:**

ACCESSING A TASK OUTSIDE ITS MASTER

DATE: June 7, 1989

NAME: Dan Stock (ACM Special Interest Group on Ada, Ada Runtime Environment Working Group)

ADDRESS: R.R. Software, Inc.
2145 Crooks Rd. #50
Troy, MI 48084

TELEPHONE NUMBER: E-mail: (ARPAnet) stockd@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 5.8, 9.2, 9.4

PROBLEM:

There is a case, specifically permitted by the language definition as interpreted by AI-00167, in which a task can be accessed from outside its master. This one anomaly causes a run-time penalty, in terms of either execution or storage overhead, and should be eliminated from the language.

SPECIFIC REQUIREMENT/SOLUTION CRITERIA:

The language should be modified to prohibit accessing a task from outside its master.

CURRENT WORKAROUNDS:

Typically, implementations represent task values by a pointer to a data structure (i.e., a task control block). This data structure is deallocated when exiting the task's master. There is one case, however, which requires special handling. Consider the following code fragment:

```
...
task type T;

function F return T is
  T1 : T;
begin
  return T1;
end F;
...

if F_TERMINATED then ...
```

In this example, the return value of function F is a task declared local to the function; consequently the returned task depends on the function as its master, and the function may not exit until the dependent task T1 has terminated. If an implementation is to reclaim the storage for a task control block upon exiting the task's master scope, such reclamation will render the pointer to the result's task control block invalid prior to return from the function. Implementations typically use one of two approaches to prevent the subsequent erroneous references to the reclaimed storage.

First, an implementation may have a task value point to an temporary, which in turn points to the task

control block. When a task terminates, the corresponding temporary is changed to point to a "dummy" task control block, which is used to represent all terminated tasks. This alternative uses extra storage for each task, and requires an extra difference for each reference to a task object. Additionally, some implementations do not recover the temporary values, leading to potential storage exhaustion.

Second, prior to deallocating a task's storage when exiting its master, implementations may introduce special code to check that the task is not used as a function return value. If the task is used as a return value, the deallocation of the task's storage is deferred (or eliminated). In general, this check must be performed at runtime, incurring performance penalties. Consider, for example, the following:

```
task type T;

T1 : T;
FOO : BOOLEAN;

function F1 return T is
    T2 : T;
    ...
    function F2 return T is
    begin
        ...
        if FOO then
            return T2;
        else
            return T1;
        end if;
    end F2;
begin
    return F2;
end F1;
```

In general, the value of FOO cannot be determined at compile time, therefore the compiler can not determine whether the local T2 or the global T1 will be returned by function F2. Consequently, the cost of determining whether the function return value is dependent on the function itself must be incurred at run-time. This runtime determination must be performed either when exiting a function returning a task type, or whenever exiting a master. In both cases, the runtime overhead will occur even if the specific (pathological) case of accessing outside the master never occurs.

Variations on the two above approaches are possible. In particular, AI-167 suggests an implementation approach; however, the AI is not clear about how examples like the last one above are to be handled. Attempts to implement the approach suggested by the AI lead inexorably to the run-time disadvantages of one or the other of the above approaches: either task information is inefficiently represented (requiring an extra level of indirection), or special code is needed for functions that return tasks.

POSSIBLE SOLUTIONS:

The simplest solution is to define returning a task outside its master as erroneous.

For additional references to Section 5. of ANSI/MIL-STD-1815A, see the following sections, revision request numbers, and revision request titles in this document.

REVISION REQUEST

<u>NUMBER</u>	<u>TITLE</u>	<u>SECTION</u>
0141	INCLUDE "WHEN" IN RAISE STATEMENT SYNTAX	11

**ADA 9X REVISION REQUESTS
THAT REFERENCE
ANSI/MIL-STD-1815A**

SECTION 6. SUBPROGRAMS

SUBPROGRAM CALLBACK

DATE: February 10, 1989

NAME: Bill Taylor (from material supplied by members of Ada-UK)

ADDRESS: Ferranti Computer Systems Limited
Ty Coch Way
Cwmbran
Gwent
NP44 7XX
UK

TELEPHONE: +44 6333 71111

ANSI/MIL-STD-1815A REFERENCE: 6.

PROBLEM:

When attempting to partition a system into a number of loosely-coupled subsystems, the situation can arise where, within a single thread of control, the control flow between two subsystems is in both directions. The alternatives are to reallocate the functionality (and compromise the modularity) or use a "call-back" facility, which requires language support.

Although not frequently encountered in the higher levels of application code, such a situation tends to occur at the lower levels of "infrastructure" software, particularly in a distributed processing environment.

One application area is Human/Computer Interaction (HCI) where the generic HCI support software needs application-specific information to enable it to validate the dialogue.

EXAMPLE:

In a menu-based dialogue sequence, there are points where the operator supplies parameter information through free-format input devices such as a keyboard. These parameters need to be dynamically checked (both syntactically and semantically) to ensure that the operator input is consistent with the operational state of the system, so as to prevent the operator wasting time continuing a menu dialogue sequence with an invalid parameter.

In developing reusable HCI software, the mechanisms of HCI need to be provided in an application independent manner. These mechanisms are tailored to a particular application through applications provided "configuration" data structures.

Semantic checking is often very application specific. The mechanisms involved depend on the activities of the application system and how information is retained with the system. Therefore, the semantic checking mechanisms must be provided in user written subprograms.

These subprograms need to be invoked from within the normal thread of control of the HCI software. The design would be considerably simplified if this could be achieved by implanting appropriate subprogram(s) as part of the configuration data structures.

The generic equivalent would be:

```
generic
  with function Is_Valid ( Identity_1: String) return Boolean;
  with function Is_Valid ( Identity_2: String) return Boolean;
  with function Is_Valid ( Identity_1: Real) return Boolean;
  with function Is_Valid ( Identity_2: Real) return Boolean;
package HCI_Support is ...
```

IMPORTANCE: IMPORTANT (almost ESSENTIAL)

Without a solution to this problem, other languages will be preferred (where the user has the choice). Programs written in Ada which require this capability will be complicated by the cumbersome apparatus of the various workarounds.

CURRENT WORKAROUNDS:

One workaround is to use generic units with formal subprograms and instantiate the generic for every actual subprogram. Whereas this approach is appropriate in many instances and indeed can be a superior solution to the use of subprogram parameters, it is not a universal solution:

- Currently, very few implementations support code sharing for generic instances efficiently.
- It may be required to call the same "subsystem" from a number of different places and with different actual subprograms.
- In a distributed system, it is sensible to physically separate client and server (e.g. the application and the HCI support).

A second workaround is to use unchecked conversion and the ADDRESS attribute. This is non-portable, potentially unreliable and presupposes that there is a mechanism to enable the server to effect a procedure call given its 'signature' and its address.

A third workaround is to return to the client with an indication of the action to be taken, following which the client makes a second call to the server (with an indication of the results of the requested action). In the HCI case, this leaves the choice as to whether to freeze the dialogue until the verification is reported or to assume verification will succeed, move onto the next menu selection, and then take appropriate (user-unfriendly) action when the client indicates otherwise. Either approach is cumbersome to implement and undesirable from a human operator viewpoint.

A fourth workaround is to move some knowledge of the client concern to the server. This approach severely compromises modularity and the potential for reuse.

POSSIBLE SOLUTIONS:

The most obvious solution is to be able to declare subprograms which can take subprograms as parameters. The matching rules would be as for generic formal subprograms.

A more general solution is to allow subprograms as types and objects, so allowing them to be embedded in record structures.

SUBPROGRAMS AS PARAMETERS

DATE: May 23, 1989

NAME: Jurgen F H Winkler

ADDRESS: Siemens AG ZFE F2 SOF3
Otto-Hahn-Ring 6
D-8000 Munchen 83
Fed Rep of Germany

TELEPHONE: +49 89 636 2173

ANSI/MIL-STD-1815A REFERENCE: 6.1, 9.5

PROBLEM:

Allow subprograms as parameters of subprograms and task entries

Since Ada allows tasks as parameters it seems reasonable enough to allow subprograms as parameters, too.

This was required in Tinman(June 76). The discussion at the Cornell Workshop (Oct. 1976) was mostly negative [FW 77:27]. After that it was no more required in Steelman.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Define a generic subprogram GS and create for each different actual parameter an instantiation of GS.

This leads to somewhat clumsy programs and results typically in code duplication despite the fact that formal subprograms have been present in programming languages since the early days of ALGOL60 and Fortran and have been usually implemented without code duplication.

POSSIBLE SOLUTIONS:

FIRST SOLUTION

This solution is very much similar to classic approaches as e.g. in Pascal or Algol. The main difference is, that the simple names of the immediate parameters of a formal subprogram are specified in the specification of this formal subprogram. This allows the named association in calls of a formal subprogram. For nonimmediate parameters of a formal subprogram no names are specified.

```
parameter_specification ::=  
    designator_list : mode formal_parameter_with_pids [ := expression]
```

```
formal_parameter_with_pids ::=  
    type_mark | formal_subprogram_with_pids
```

```
formal_subprogram_with_pids ::=  
    procedure [parameter_profile_with_pids]
```

```

    | function [parameter_profile_with_pids] return type_mark

parameter_profile_with_pids ::=
    (parameter_indication_with_pids { ;parameter_indication_with_pids})

parameter_indication_with_pids ::=
    designator_list : parameter_indication

parameter_indication ::=
    mode parameter_type [:= <>]

parameter_type ::=
    type_mark | formal_subprogram

formal_subprogram ::=
    procedure [parameter_profile]
    | function [parameter_profile] return type_mark

parameter_profile ::=
    (parameter_indication {;parameter_indication})

```

SECOND SOLUTION

In this solution the names of all parameters (direct and indirect) of formal subprograms are specified. This leads to a simpler syntax, but to superfluous names which are meaningless, and which disturb the writer as well as the reading of a program. The same holds for superfluous defaults expressions.

```

parameter_specification ::=
    designator_list : mode formal_parameter_with_pids [:= expression]

formal_parameter_with_pids ::=
    type_mark | formal_subprogram_with_pids

formal_subprogram_with_pids ::=
    procedure [formal_part]
    | function [formal_part] return type_mark

```

THIRD SOLUTION

This solution has an even simpler syntax, but show different formats for formal parameters:

```

identifier:mode type_mark                vs.
procedure identifier [formal_part]

```

```

parameter_specification ::=
    identifier_list : mode type_mark [:=expression]
    | subprogram_specification

```

FOURTH SOLUTION

In this solution subprogram types are introduced analogously to task types, which are already present in Ada (they have been introduced after the June 1979 version). This solution seems to be closest to the spirit of Ada, which includes already task types.

```
subprogram_specification ::=  
    procedure [type] identifier [formal_part]  
    | function [type] identifier [formal_part] return type_mark
```

But the solution can not exactly parallel that for task types. The purpose of a task type in Ada is the creation of several incarnations of one task description (specification + body). This possibility is of no great importance for subprograms because a subprogram can be activated several times at one point in time (either by recursion or by a number of different tasks).

For subprograms, especially as formal parameters, it is important that several different subprograms (bodies) can be defined for the same interface (= specification). This can be accomplished either by conformance rules for subprogram specifications or by the following additional rule for subprogram body:

subprogram body:

```
subprogram body ::=  
    designator : subtype_indication is  
        [declarative_part]  
    begin  
        sequence_of_statements  
    [ exception  
        exception_handler  
        {exception_handler}]  
    end [designator];
```

If subprogram types are limited no further problems arise. If they are only private the rules for assignment must be defined in a way that excludes the assignation of local subprograms to global variables. It seems that some form of assignation is necessary, because without assignation it is e.g. not possible to have a tree with different subprograms (of one type) at different nodes.

SUBPROGRAM SPECIFICATION

DATE: January 15, 1989

NAME: William Thomas Wolfe

ADDRESS: Home: 102 Edgewood Avenue #2
Clemson, SC 29631 USA

Office: Department of Computer Science
Clemson University
Clemson, SC 29634 USA

TELEPHONE: Home: (803) 654-7030
Office: (803) 656-2847
E-mail: wtwolfe@hubcap.clemson.edu

ANSI/MIL-STD-1815A REFERENCE: 6.1(3)

PROBLEM:

The idea of a subprogram specification is incomplete in that although parameter structures are specified, the question of references to nonlocal objects which are not parameters is not addressed.

CONSEQUENCES:

Software debugging and maintenance is complicated by the fact that one can never really be certain precisely which nonlocal objects a given subprogram is reading or modifying. Also, there is no way for the designer of a subprogram to guarantee that the caller of the subprogram will not create a form of aliasing whereby a nonlocal object which is referenced by the subprogram is also supplied as an actual parameter, thus leading to the incorrect functioning of the subprogram.

CURRENT WORKAROUNDS:

Comments can be used instead, but the compiler cannot enforce either the presence of the comments or the extent to which the comments accurately reflect the subprogram's actual behavior.

POSSIBLE SOLUTIONS:

Add to the idea of a subprogram specification a facility whereby references to nonlocal objects can be enumerated in such a way that if the compiler detects an attempt by the subprogram to reference an undeclared nonlocal object, an error message will be generated. An error message should also be generated if a caller of a given subprogram attempts to supply one of the nonlocal objects which is being referenced by the subprogram as an actual parameter.

The facility for declaring nonlocal objects should be similar to the parameter declaration facility, up to and including the specification of "in", "out", and "in out" modes. As with parameters, attempts by the subprogram to violate the mode specification of a nonlocal object should cause compilers to generate the appropriate error messages.

READING OF OUT PARAMETERS

DATE: October 25, 1988

NAME: S. Tucker Taft

ADDRESS: Intermetrics, Inc.
733 Concord Avenue
Cambridge, MA 02138

TELEPHONE: (617) 661-1840
E-mail: stt@inmet.inmet.com
E-mail: uunet!inmet!stt

ANSI/MIL-STD-1815A REFERENCE: 6.2(5)

PROBLEM:

There seems no reason to disallow reading of OUT parameters. Reading an OUT parameter prior to its initialization is exactly equivalent to reading a local variable prior to its initialization, and hence does not represent a new erroneous condition in the language. On the other hand, it is a frequent situation where an OUT parameter is initialized early in a subprogram, and then ideally would be available for reading later. There have been untold number of bugs and excess code introduced by trying to work-around this limitation by adding local variables to hold the value until just prior to the return statement, at which time it is copied into the OUT parameter.

IMPORTANCE:

Additional code and additional bugs will be created to work-around an arbitrary restriction.

CURRENT WORKAROUNDS:

Here is a typical example of what the programmer would *like* to write:

```
procedure Sum(X : Vector; Result : out Integer) is
begin
  Result := 0;
  for I in X'Range loop
    Result := Result + X(I);
  end loop;
end Sum;
```

Here is what they end up writing:

```
procedure Sum(X : Vector; Result : out Integer) is
  Local_Result : Integer := 0;
begin
  for I in X'Range loop
    Local_Result := Local_Result + X(I);
  end loop;
  Result := Local_Result; -- This statement is frequently
```

```
-- forgotten, especially if there are multiple returns  
-- from the subprogram  
end Sum;
```

POSSIBLE SOLUTIONS:

Simply remove the current restriction, and allow the reading of OUT parameters, and state that it is erroneous to read an OUT parameter prior to its initialization. This should simplify compilers, rather than make them more complicated.

SUBPROGRAM BODIES AS GENERIC INSTANTIATIONS**DATE:** January 31, 1989**NAME:** David Brookman**ADDRESS:** Magnavox Electronic Systems Company
1313 Production Road
Department 542
Fort Wayne, IN 46808**TELEPHONE:** (219) 429-4440**ANSI/MIL-STD-1815A REFERENCE:** 6.3**PROBLEM:**

The body to a subprogram cannot be specified directly as the instantiation of a generic, or as a renaming of some other subprogram. Frequently subprograms are declared in a package specification. In the package body, it would be desirable to implement the subprogram body as a generic instantiation.

IMPORTANCE: ADMINISTRATIVE**CURRENT WORKAROUNDS:**

The subprogram body may contain only a call to the desired generic. This works, but it complicates the code. Also it may cause a loss in efficiency since some compilers will not optimize away the unnecessary subprogram call.

example:

```
function Record_Exists
(
  Key : in Key_Type
) return Boolean is
begin -- Record_Exists
  return Our_List_Package.Record_Exists
    (Key => Key);
end Record_Exists;
```

POSSIBLE SOLUTIONS:

Provide the ability to give the body of the subprogram via a renaming clause. The parameter profile must match between the new subprogram and the one being referenced. However this may be a problem, since often the parameters are a derived types of the parameters in the subprogram being referenced. Although the following code is legal by itself, it is not legal in the context of supplying a proper body for a subprogram which has been previously specified.

example:

```
function Record_Exists
```

```
(  
Key : in Key_Type  
) return Boolean renames Our_List_Package.Record_Exists;
```

PRAGMA SELECTIVE INLINE**DATE:** April 4, 1989**NAME:** Eric F. Heck**ADDRESS:** GE Aerospace GCSD
Mail Stop 13-7-5
Camden, NJ 08102**TELEPHONE:** (609) 338-4776**ANSI/MIL-STD-1815A REFERENCE:** 6.3.2(1)**PROBLEM:**

The ability to selectively inline a routine is needed in embedded real-time systems where an interrupt service routine (ISR) (interrupt task) calls a routine that application code also calls. In the ISR, the run-time system overhead in making the call is unacceptable, therefore the routine needs to be inline. In the application code, the system can tolerate the run-time system overhead in making the call, therefore the routine need not be made inline here. Of course, the routine could be made inline everywhere it is called (via Pragma Inline); however this causes the amount of ROMable code to increase and can lead to sizing problems.

IMPORTANCE: ESSENTIAL

This request is essential to real-time embedded systems. The efficient use of interrupt tasks depends on this request being implemented.

CURRENT WORKAROUNDS:

ISRs are written in assembly language.

POSSIBLE SOLUTIONS:

When a routine is declared using Pragma Inline, every place where it is called, the routine is physically placed inline. However, the original source routine is available after a system is linked and located (the routine is in the load map), although that code is never executed. Pragma Selective Inline could make use of this source code. If a routine is defined using Pragma Selective Inline, then those who call the routine would determine if the routine should be "inlined" or not. The default situation would be to make the routine inline.

EXECUTION OF A PROGRAM UNIT BY ITS ADDRESS

DATE: October 15, 1988

NAME: Richard Bryant

ADDRESS: MS: E301
Delco Systems Operation
6767 Hollister Ave.
Santa Barbara, CA 93117

TELEPHONE: (805) 961-7601

ANSI/MIL-STD-1815A REFERENCE: 6.4

PROBLEM:

Ada provides the capability to take the address of any program unit (LRM 13.5, 13.7.2) but the language does not provide any corresponding capability to perform actions with the object's address. Most embedded systems require the ability to execute separately compiled units (Built In Test and low level input-output functions) that reside in ROM. Access to these units is generally through a procedure jump table. It would be desirable for Ada to allow the execution of these units by their address from within the Ada language.

SPECIFIC REQUIREMENTS/SOLUTION CRITERIA:

Extend the operations that can be performed on an address object of a subprogram unit to include calling the subprogram unit by its address.

Such a capability would include the passing of parameters and receiving return values from subprogram units. By requiring a defined access type for the subprogram unit address the compiler can check parameters' type and value statically thereby insuring correct usage.

IMPORTANCE:

If this change is not supported, implementers will probably decide to use another language ("C", LISP, ASSEMBLY), depending on the application, to perform those functions that require this feature. And this may be seen as a another reason why Ada can not support embedded applications.

CURRENT WORKAROUNDS:

Avionic systems require hardware vendor supplied Built-In-Test (BIT) software and other ROM based module specific functions to be executed by the operational flight software. Currently it is not possible to execute these functions from within Ada even though their addresses are known because they are considered to be outside of the Ada program.

There are several situations where such a need exists. Embedded systems frequently have code stored in ROM. Two such instances are the Built-In-Test functions shipped in the hardware module by the vender, and, A standardized ROM BIOS containing input-output device interface code. A ROM BIOS is currently being proposed by JIAWG members as a way of obtaining operating system/runtime and application code portability across different hardware vendors Common Application Processor (CAP) modules. In both cases

The ROM code is outside of the link-time Ada program but must be accessed by the Ada program at runtime.

Another kind of application domain where subprogram execution by address is needed are applications that perform dynamic linking at runtime. Some embedded systems are highly memory constrained for their application size due to power/weight/space/processor architecture or other operational environmental reasons. Many of these systems use memory overlays to over-come these constraints. In this environment, the use of subprogram addresses to resolve external references during dynamic loading and linking offer a large pay back in reduced code complexity and size.

The only way to execute these functions now is to create a package that defines assembly language routines that indirectly call the target subprogram using pragma INTERFACE. This method adds another layer of runtime overhead. And since parameter checking can not be performed by the compiler on the indirect call this process is also error-prone.

Another approach might be to use pragma INTERFACE directly, but the pragma only allows the linking of subprograms written in another language to an Ada program; it does not address the problem of calling subprograms that are external to an Ada program. Moreover, the pragma INTERFACE capability need not be provided by all implementations (LRM 13.9).

For similar reasons 'generics' are also not applicable. Besides having the same syntactic limitations as a library subprogram, they also are part of the linkable image.

POSSIBLE SOLUTIONS:

Here are two possible implementations for subprogram address type definition and actual call usage.

```
1)
-- Data type
procedure PERFORM ( ARG1 : in integer; ARG2 : in address );
type PROC_ACCESS is access procedure;

-- Data object description and initialization
PROC : array (0..15) of PROC_ACCESS;
PROC(3) := PERFORM'ADDRESS;

-- Actual call
PROC(3) (X, ADDR);
```

This implementation is the most general but it has one problem, there is no way for the compiler to statically check the correctness of the actual parameter list. The second implementation solves that problem by providing more specific access type checking.

```
2)
-- Data type definitions
procedure READ ( ARG1 : in integer; ARG2 : in address );

function WRITE ( ARG1 : in integer; ARG2 : in address ) return boolean;
for WRITE use at 16#0020#;

type READ_ACCESS is access procedure READ;
type WRITE_ACCESS is access function WRITE;
```

```
type PROC_TABLE is record
  P1 : READ_ACCESS;
  P2 : WRITE_ACCESS;
end record;

-- Data object description and initialization
CALL_TABLE : PROC_TABLE;
CALL_TABLE.P1 := READ_ADDRESS;
CALL_TABLE.P2 := WRITE_ADDRESS;

-- Actual call
CALL_TABLE.P1 (X, ADDR);
RESULT := CALL_TABLE.P2 (X, ADDR);
```

This implementation provides sufficient information to insure that procedure parameters and calling conventions are enforced by the compiler, and provides for the efficient execution of external subprogram units from within Ada.

DIFFICULTIES TO BE CONSIDERED:

There does not seem to be any technical difficulties in implementing this feature. This feature appears to be contrary to the spirit of some Algol based languages from which Ada was derived. But Ada contains many additional features that are outside of the traditional Algol language definition. Therefore, adherence to Algol should not be an acceptance criteria.

EXPLICIT INVOCATION OF DEFAULT PARAMETER

DATE: March 22, 1989

NAME: E.N. Thomas

DISCLAIMER: The views expressed in this note are those of the author, and do not necessarily represent those of SD-Scicon PLC.

ADDRESS: SD-Scicon PLC
Pembroke House
Pembroke Broadway
CAMBERLEY
Surrey
UK
GU15 3XD

TELEPHONE: +44 276 686200

ANSI/MIL-STD-1815A REFERENCE: 6.4.2

PROBLEM:

Section 6.4.2 provides the facility expressions for parameters, which are invoked at the calls by omitting the respective parameter association.

Where the visible subprograms contain a mixture of subprograms with default parameters and overloaded subprograms, readability of the source is often compromised by the inability to indicate that a default is being deliberately invoked at any given call. This is particularly true if the source style is to use named associations exclusively, since no named association can be given for parameters where the default value is required.

IMPORTANCE: ESSENTIAL

Compromises source code readability in large software modules, leading to maintenance problems if it is required to change the number and nature of the default parameters, or change sets of overloaded subprograms.

CURRENT WORKAROUNDS:

Some improvement can be made by the use of explanatory comments, but this will not provide any compile time support for inconsistency arising from the sort of changes listed above.

POSSIBLE SOLUTIONS:

The syntax for parameter associations at subprogram calls should be extended to permit an explicit "default" both for named and positional associations. In order to avoid the introduction of a new reserved word, it is suggested that the box compound delimiter of 2.2(6) should be used. The syntax in 6.4(2) for actual parameter thus becomes as follows.

```
actual_parameter ::=  
    <<as existing, followed by>>  
    | <>
```

This box would be acceptable where there was a default, in the same way that omission of the parameter association is only acceptable in such cases. As described above, the box could be used for both named and positional association.

Some choice remains as to whether or not to continue to allow implicit use of defaults by omission of parameter associations, particularly so that existing Ada programs can avoid the need for widespread changes.

- One view is to insist that if named associations are used then all parameters must be given-but this would require changes to existing source.
- An alternative is to provide information at the subprogram declaration to constrain the allowed forms of calls. This could take the form of a pragma citing the subprogram name and indicating the form of calls, possibly:

```
pragma CALLS (TO => CREATE, DEFAULT => EXPLICIT);
```

Such a pragma would be subject to placement restrictions equivalent to the present pragma `INLINE`. The values for its `DEFAULT` argument might include `EXPLICIT`, `IMPLICIT`, and `ANY`; further, separate control might be given for positional and named associations. (`EXPLICIT` would require use of box for all default parameters, `IMPLICIT` would supply present Ada rules, and `ANY` would allow any mixtures of the two.)

Instead of a program, it might be possible to adapt the syntax of the procedure declaration to allow specification of these constraints.

MODE OF PARAMETERS OF A FUNCTION

DATE: January 14, 1989

NAME: William Thomas Wolfe

ADDRESS: Home: 102 Edgewood Avenue #2
Clemson, SC 29631 USA

Office: Department of Computer Science
Clemson University
Clemson, SC 29634 USA

TELEPHONE: Home: (803) 654-7030
Office: (803) 656-2847
E-mail: wtwolfe@hubcap.clemson.edu

ANSI/MIL-STD-1815A REFERENCE: 6.5(1)

PROBLEM:

The parameters of a function subprogram must have the mode "in".

CONSEQUENCES:

Functions for which the user would reasonably expect the parameter(s) to be modified cannot be implemented.

As a first example, consider function SQUEEZE (JUICE_SOURCE : in out FRUIT) return JUICE;

Clearly, the caller of such a procedure would reasonably expect that the state of the JUICE_SOURCE would be modified.

Consider also the example of a B_PLUS_TREE whose abstraction is defined to include the concept of a current key. When a query is made via function KEY_EXISTS (IN_B_PLUS_TREE : in out B_PLUS_TREE; DESIRED_KEY : in KEY_TYPE) return BOOLEAN;

The caller of this function expects that the current key of the tree which was passed as a parameter will be equal to the DESIRED_KEY if the function returns true, and unchanged otherwise.

CURRENT WORKAROUNDS:

The function can be defined such that a pointer to the "real parameter" is passed instead. In the case of the second example, this would require that the implementation of a B_PLUS_TPEE begin with a pointer to the B_PLUS_TREE's descriptor, rather than with the descriptor itself. A consequence of this is that every operation on the B_PLUS_TREE will have to dereference this pointer, which results in a language-imposed inefficiency in the implementation -- a waste of both time and space.

POSSIBLE SOLUTIONS:

Modify the sentence "The specification of a function starts with the reserved word "function", and the

parameters, if any, must have the mode "in" (whether this mode is specified explicitly or implicitly).", in ANSI/MIL-STD-1815A 6.5 (1), to read "The specification of a function starts with the reserved word "function", and the default mode for the parameters, if any, is "in".".

ALLOW OVERLOADING OF "="**DATE:** October 28, 1988**NAME:** S. Tucker Taft**ADDRESS:** Intermetrics, Inc.
733 Concord Avenue
Cambridge, MA 02138**TELEPHONE:** (617) 661-1840
E-mail: stt@inmet.inmet.com
E-mail: uunet!inmet!stt**ANSI/MIL-STD-1815A REFERENCE:** 6.7(4)**PROBLEM:**

The "=" operator should be definable on any type, so long as the result type is Standard. Boolean (so "/=" is well defined). It is currently possible to accomplish the same thing by a clever use of generics and renaming.

There are many times when the full restrictions of limited types are undesirable, while it would still be very useful to redefine the semantics of "=" (e.g., for rational numbers, equality means the numerator and denominator have the same ratio, not that they be identical).

IMPORTANCE:

The "John Goodenough" trick for defining "=" will begin to appear all over the place, or users will simply suffer quietly under the existing restriction.

CURRENT WORKAROUNDS:

The original rationale for disallowing the hiding of pre-defined "=" by a user-defined operator is probably that when used as a component the predefined "=" is used for the enclosing object. However, there is good precedent that predefined operators reemerge in certain circumstances, such as the definition of membership, without precluding their explicit hiding.

The "John Goodenough" trick for defining "=" is:

```

generic
  type Operand is limited private;
  with function Equal(Left, Right : Operand) return Boolean
is <>;
package Redefine_Equal is
  function "="(Left, Right : Operand) return Boolean
    renames Equal;
end Redefine_Equal;

```

...

```
package My_Equal is new Redefine_Equal(Non_Limited_Type,  
Equal);  
function "="(Left, Right : Non_Limited_Type) return Boolean  
renames My_Equal."=";
```

Hence, it is very possible, but absurdly round-about.

POSSIBLE SOLUTIONS:

Simply revise the current restriction of redefining "=" to specify only that the parameters be the same type, and that the result be the predefined type BOOLEAN.

OVERLOADING "="**DATE:** January 14, 1989**NAME:** William Thomas Wolfe**ADDRESS:** Home: 102 Edgewood Avenue #2
Clemson, SC 29631 USAOffice: Department of Computer Science
Clemson University
Clemson, SC 29634 USA**TELEPHONE:** Home: (803) 654-7030
Office: (803) 656-2847
E-mail: wtwolfe@hubcap.clemson.edu**ANSI/MIL-STD-1815A REFERENCE:** 6.7(4)**PROBLEM:**

It is not possible to overload "=" with parameters of different types.

CONSEQUENCES:

This prevents the overloading of "=" between the predefined type `STRING` and the user-defined type `VARIABLE_LENGTH_STRING`, preventing the implementation of a *variable-length-string* package which provides all the usual operators. Only the "=" between `VARIABLE_LENGTH_STRING` and `STRING` cannot be provided; all other operators between the two types can presently be provided.

More generally, any two types which one would reasonably expect to be compatible (bounded versus unbounded containers, for example) cannot presently be made compatible with respect to the "=" operator.

CURRENT WORKAROUNDS:**POSSIBLE SOLUTIONS:**

Delete the sentence "The explicit declaration of a function that overloads the equality operator "=", other than by a renaming declaration, is only allowed if both parameters are of the same limited type." from ANSI/MIL-STD-1815A 6.7 (4).

For additional references to Section 6. of ANSI/MIL-STD-1815A, see the following sections, revision request numbers, and revision request titles in this document.

REVISION REQUEST

<u>NUMBER</u>	<u>TITLE</u>	<u>SECTION</u>
0117	PRE-ELABORATOR	3
0094	IDENTIFIER LISTS AND THE EQUIVALENCE OF SINGLE AND MULTIPLE DECLARATIONS	3
0111	FAULT TOLERANCE	5
0096	LIMITATION ON USE OF RENAMING	8
0101	IMPLEMENTATION OF EXCEPTIONS AS TYPES	11

**ADA 9X REVISION REQUESTS
THAT REFERENCE
ANSI/MIL-STD-1815A**

SECTION 7. PACKAGES

COMPILATION UNITS**DATE:** April 21, 1989**NAME:** Deb Lieberman**ADDRESS:** Texas Instruments
P.O. Box 869305 MS 8435
Plano, TX 75086**TELEPHONE:** (214) 575-3516**ANSI/MIL-STD-1815A REFERENCE:** 7.**PROBLEM:**

It is not possible for the visible part of the package specification to declare an object that is of private type or has a component that is of private type.

IMPORTANCE: IMPORTANT**CURRENT WORKAROUNDS:**

An "extra" package that declares the necessary objects is created or the private type is removed.

POSSIBLE SOLUTIONS:

Allow a package to have three separate compilation units:

- (1) private part
- (2) specification (visible) part
- (3) body part

PROBLEMS WITH OBJECT ORIENTED SIMULATION**DATE:** June 29, 1989**NAME:** Kjell Rose**ADDRESS:** NDRE
P O Box 25
Kjeller N2007
Norway**TELEPHONE:** 011 47 6 807454**ANSI/MIL-STD-1815A REFERENCE:** 7., 9.**PROBLEM:**

In order to gain practical experience with Ada, we decided to implement a program for simulation of a sea-invasion scenario with multiple vessels, anti-shipping missiles, surface-to-air missiles and guns, in Ada.

During a study of future anti-shipping missiles, we had difficulties in describing alternate missile configurations in Ada.

After a (re)discovery of object oriented programming, we concluded that Ada's facilities for modularization (packages and tasks) are not sufficient for our future needs.

IMPORTANCE: ESSENTIAL**CURRENT WORKAROUNDS:**

The simulation program was translated to C++, and the development of the program has since continued without problems.

POSSIBLE SOLUTIONS:

Ada 9X should support object oriented programming. Classes should be included.

Rationale for Design of the Ada Programming Language states in chapter 8.1:

Facilities for modularization have appeared in many languages. Some of them such as Simula, Clu, and Alphard provide dynamic facilities which may entail expensive run-time overhead.

The C++ language has shown that classes may be implemented effectively and without garbage collection.

MUTUAL VISIBILITY REGION**DATE:** July 30, 1989**NAME:** William Thomas Wolfe**ADDRESS:** Home: 102 Edgewood Avenue #2
Clemson, SC 29631 USAOffice: Department of Computer Science
Clemson University
Clemson, SC 29634 USA**TELEPHONE:** Home: (803) 654-7030
Office: (803) 656-2847
E-mail: wtwolfe@hubcap.clemson.edu**ANSI/MIL-STD-1815A REFERENCE:** 7.

(This request supercedes a previous request regarding declaration groupings, whose revision request number is somewhere in the range 8901160025-8901160035; that request is hereby withdrawn)

PROBLEM:

Many situations arise in software construction which are naturally modelled by a region in which all declared entities are mutually visible, particularly situations in which a number of tasks need to be mutually visible in order to call each other as necessary. Doing this in Ada is very awkward at present; it is necessary to have an initialization entry in which some surrounding unit passes pointers to each neighboring task, or to use some sort of mailbox system, etc.; none of these solutions possesses the clarity of simply being able to declare a "mutual visibility region" so as to directly model the situation.

CONSEQUENCES:

Programmer time is wasted, and program clarity is reduced.

CURRENT WORKAROUNDS:

As described above.

POSSIBLE SOLUTIONS:

Provide an alternate form of the package which has the property that all declared entities are mutually visible.

CONTROL OVER VISIBILITY OF TASK ENTRIES

DATE: March 22, 1989

NAME: K.M. Farrington (from material originally discussed in Nissen and Wallis (1984))

ADDRESS: SEMA Group (UK)
Orion Court
Forbury Park
Kenavon Drive
READING
RG1 3DG
UK

TELEPHONE: +44 734 508961

ANSI/MIL-STD-1815A REFERENCE: 7.1, 9.1

PROBLEM:

The notion of programs as collections of services and service users can be conveniently represented through the use of the package. Furthermore the package encourages the isolation of the definition of a service interface from the details of its particular implementation. In the case of packages which contain tasks, this ability for information hiding is too inflexible.

Tasks may be declared at three places in a package:

- (i) - in the visible part of the specification
- (ii) - in the private part of the specification
- (iii) - in the declarative part of the body

In (i), all entries of the task are available to the package user. In (ii) and (iii), all entries are invisible; indeed the package user may not even be aware that tasks are present.

In the case where task entries form part of the service interface provided by a package, it cannot be assumed that all entries in the task are intended for use by the package user. Some entries may be for internal control or communication, particularly where the service is implemented by several co-operating tasks. It is not possible to selectively reveal the 'user' entries.

The example in Nissen and Wallis serves to illustrate this problem. A package is defined which provides a disk writing service. This is implemented in a buffered fashion. As a result the package contains two tasks:

- Task A carries out the actual writing to disk
- Task B, a buffering process, accepts write requests from the service user and read requests from Task A

Task B must be declared with two entries, but only one, the write interface needs to be visible to the user.

As a further example, tasks in a package are given initialization entry points, by which they can synchronize

with each other at program startup. The intention is that these should be called only once, by the sequence of statements in the package body itself. These entry points should not be visible to the package user as incorrect use may cause synchronization problems.

IMPORTANCE: IMPORTANT

The lack of control over the visibility of entries contrasts with the other information hiding capabilities of packages.

CURRENT WORKAROUNDS:

Nissen and Wallis suggest: invoking each 'user' entry through a procedure. These procedures are part of the visibility interface and allows the task to be hidden completely in the package body.

Alternatively, instead of having a procedure in the package body to call the entry, the procedure declaration could represent a renaming of the entry.

POSSIBLE SOLUTIONS:

A semantic change which allows a task declaration for the same task to appear in both the visible part of a package specification and either the private part or the package specification or the declarative part of the package body, i.e., in terms of i, ii, and iii above, i and (ii or iii).

Two possible interpretations of this situation could be used:

- (a) the set of entries for the task consist of the union of the two sets of entries given in the two specifications, of which only those given in the visible part are available outside the package.
- (b) the set of entries for the task consist only of the set specified at place ii or iii, of which the set at i must be a subset and are the only ones visible outside the package.

(a) would allow the current semantics to remain valid. (b) would give rise to more defensive programs.

Many syntactic solutions, such as allowing entries to be declared as private, or having a private part in a task declaration, can be rejected on the grounds that these are needed only for task declarations appearing in package specifications. There are many other places where task declarations may appear.

(SOURCE: Section 9.5 of Style Guide in: Nissen and Wallis (eds.) (1984)

CONSTANTS DEFERRED TO PACKAGE BODY**DATE:** March 22, 1989**NAME:** E.N. Thomas**DISCLAIMER:** The views expressed in this note are those of the author, and do not necessarily represent those of SD-Scicon PLC.**ADDRESS:** SD-Scicon PLC
Pembroke House
Pembroke Broadway
CAMBERLEY
Surrey
UK
GU15 3XD**TELEPHONE:** +44 276 686200**ANSI/MIL-STD-1815A REFERENCE:** 7.4, 7.4.3, 3.1, 3.8.1**PROBLEM:**

The present Ada language provides the ability to declare deferred constants, but only of a private type, and the full declaration of the deferred constant must appear in the private part of the package (specification). Further, the language includes the concept of incomplete type declarations, which in some cases can have their corresponding full type declaration occurring in the package body when the incomplete declaration occurred in the package specification.

There is no provision for a deferred constant to have its full declaration in the corresponding package body, and only a constant of a private type can be deferred. Incomplete types whose full declaration appears in the package body are not strictly private, although their incomplete declaration appears in the private part, so their constants cannot be deferred at all.

Although the above has been introduced in terms of constants of incomplete types whose full declaration appears in the package body, the ability to defer supplying the value of the constant until the package body is also needed for visible constants of any type, in order to allow such a value to be changed without necessarily making users of the package obsolete. The actual value of the constant could be changed by recompiling the body only, without the need to recompile all users of the package specification. (Extra dependency could be introduced in a similar way to that produced by inlining of subprogram bodies (10.3(7)), by compiling with a suitable level of optimization -- under the control of the compiler user of course).

IMPORTANCE: IMPORTANT

This is a significant missing functionality for implementing packages with visible objects.

CURRENT WORKAROUNDS:

Where a visible constant is required that can have its value postponed to the package body, the effect can be simulated by a visible function that returns the relevant value -- possibly from a hidden constant in the

package body. However, use of such a function is not equivalent in terms of the rules of overloading, when several packages have such visible functions. The impact of the function in static expressions may also be different from that of a constant.

POSSIBLE SOLUTIONS:

Provide the ability for a deferred constant to have its value supplied in the package body, and allow this to be used for constants of any type. This should preferably be done using a syntax that declares at the original deferred constant declaration that it is required to defer it to the body, so that mistakes can be detected at compile time. A possible syntax might be as follows, which allows the existing syntax to continue to support the existing semantics:

```
deferred_constant_declaration ::=
    identifier_list : constant type_mark [deferred_place];

deferred_place ::=
    private | body
```

As for existing deferred constant declarations it would be erroneous to use the value of the constant before the elaboration of the full declaration (7.4.3(4)), whether this use occurred within the package body, or within another program unit.

An extra related feature would be to allow the "constant" to be read-only when viewed from outside its package body, but writable from inside the body, subsequent to its full declaration. This ideally should have a further distinct syntax for the deferred constant declaration.

PRIVATE TYPE DERIVED FROM DISCRIMINATED TYPE

DATE: October 28, 1988

NAME: S. Tucker Taft

ADDRESS: Intermetrics, Inc.
733 Concord Avenue
Cambridge, MA 02138

TELEPHONE: (617) 661-1840
E-mail: stt@inmet.inmet.com
E-mail: uunet!inmet!stt

ANSI/MIL-STD-1815A REFERENCE: 7.4.1(3)

PROBLEM:

The full definition of a private or incomplete type with discriminants should be allowed to be a derived type, providing that the parent type has discriminants which somehow match (either by strict conformance, or more relaxed as with generic actual/formal matching of discriminated types).

It seems like a clear oversight in the language that discriminated private/incomplete types cannot be defined by derivation, and would be a very useful addition. Frequently there is a desire to have a non-private type which is equivalent to the private type, and a conversion function between them. As it is now, there is no efficient way to implement the private type in terms of the non-private type if the types are discriminated.

IMPORTANCE:

The usefulness of private types will be reduced if they cannot be defined by derivation from a discriminated type, and therefore will not be used to their full potential.

CURRENT WORKAROUNDS:

An approximate equivalence can be accomplished now by having the full definition be a record enclosing the desired parent type as the single non-discriminant component, though this almost certainly introduces significant additional overhead and complexity.

The current situation seems like an arbitrary restriction, and adds nothing to the safety of the language.

Derived types are the major mechanism in Ada for creating type hierarchies. There should be no restrictions on the ability to combine type derivation with the private/incomplete type concept.

POSSIBLE SOLUTIONS:

Explicitly allow the full definition of a private/incomplete type with discriminants to be a derived type definition. The subtypes of the discriminants must be required to conform, or constraint checks analogous to those performed at generic instantiation must be required at the point of the full type definition.

There seems to be no reason why the discriminant names or defaults from the private type definition need match those of the parent type.

USER DEFINED ASSIGNMENT

DATE: February 9, 1989

NAME: J G P Barnes (from material supplied by members of Ada-UK)

ADDRESS: Alslys Ltd
Newtown Road
Henley-on-Thames
Oxon
RG9 1EN
UK

TELEPHONE: +44-491-579090

ANSI/MIL-STD-1815A REFERENCE: 7.4.4, 4.5.2

PROBLEM:

The language does not allow the user to define assignment even for limited types. There is a lack of symmetry with regard to equality which can be redefined for such types. This leads to problems of natural expressiveness with abstract data types.

There are many occasions where predefined equality is inappropriate for an abstract data type and it is then necessary to make the type limited so that equality can be defined explicitly. However, this then makes assignment unavailable for the type. In some situations the absence of assignment is an important property of limited types. On the other hand there are circumstances where assignment is desirable. The user is then forced to declare an "assign" procedure which results in clumsy and opaque notation rather than the natural syntax `V:=E;` to balance the equality operator `"=`".

Note that in such cases the user will often simply want to make predefined assignment available but there will be situations where a more structured "operation" is required.

IMPORTANCE: IMPORTANT

This is important in a formal sense especially bearing in mind the importance currently attached to design methods, Object Oriented Programming and so on. It will overcome a view that Ada is not adequate in this area and remove an objection to treating Ada with respect in educational establishments. It might be of great value in the simplification of Ada program generators which are likely to be of growing importance in the future.

On the other hand it will not affect program performance. However, it should equally be trivial to implement.

This change would be good value.

CURRENT WORKAROUNDS:

The user has to declare a procedure such as

`procedure ASSIGN(LEFT: in out ADT; RIGHT: in ADT) is`


```
begin
    LEFT:=RIGHT;
end;
```

in the declaring package for the limited type ADT. Outside that package we then have to write

```
ASSIGN(X, Y);
```

rather than the more natural

```
X:=Y;
```

POSSIBLE SOLUTIONS:

Simply allow assignment to be defined for a limited type thus

```
procedure ":=" (LEFT: in out ADT; RIGHT: in ADT) is
...
```

within the defining package. Such a procedure would be required to have two parameters of a limited type with modes in out and in respectively. The formal names seem irrelevant. (Perhaps the first parameter could be allowed to have mode out or in out.)

The semantics could simply be that the effect of `X:=Y;` is precisely that of `ASSIGN(X, Y);` where `ASSIGN` has the equivalent body. However, it might be better to write the LRM in different terms in which case some care might be required over the properties of basic operations versus predefined operations.

The declaration of such a procedure should permit initialization of limited types.

Overload resolution would be required as for other overloaded operators.

There would be other consequences such as allowing `":="` as a generic formal subprogram.

However, this proposal does not imply any alteration to the subprogram parameter mechanism.

LIMITED TYPES TOO LIMITED**DATE:** October 10, 1988**NAME:** S. Tucker Taft**ADDRESS:** Intermetrics, Inc.
733 Concord Avenue
Cambridge, MA 02138**TELEPHONE:** (617) 661-1840
E-mail: stt@inmet.inmet.com
E-mail: uunet!inmet!stt**ANSI/MIL-STD-1815A REFERENCE:** 7.4.4(6)**PROBLEM:**

Limited types are useful any time the representation of a type requires the use of indirection, so that predefined assignment/equality would not be meaningful. However, the restrictions associated with limited types are unnecessarily harsh, and the typical result is that the types are made non-limited because of other requirements which override the protection requirements.

At the very least, it seems that initialized constants and aggregates (presuming not private) ought to be allowed for limited types. Because it is legal to return a limited type, it is possible to create the equivalent of an initialized constant using a pair of subprogram calls, so it seems clear that the restrictions are not in fact providing any additional protection, and are simply arbitrary limitations.

IMPORTANCE:

If the functionality of limited types is not beefed up, then they will continue to be of little practical use to Ada programmers, despite the additional protection they potentially provide.

CURRENT WORKAROUNDS:

The following is currently illegal (though the purpose of this revision request is to make it legal):

```
declare
  X : constant Lim_Type := Y;
begin
  <stmts>
end;
```

However, it is equivalent, in terms of required copying of the value of Y, to:

```
declare
  function Identity(L : Lim_Type) return Lim_Type is
  begin return L; end Identity;

  procedure Stmts(X : Lim_Type := Identity(Y)) is
  begin <stmts> end Stmts;
```

```
begin  
  Stmt;  
end;
```

Hence, there seems no reason to disallow the initialization of constants of limited type. Once this is recognized, there seems no apparent reason to disallow the use of aggregates for a limited type, since an aggregate is simply an initialized constant.

It is also arguable that initialized variables/allocators of limited type should be supported, given that predefined copying is already being provided as part of the return statement. However, it is harder to prove that this would create no new security "holes" for limited types.

POSSIBLE SOLUTIONS:

Simply remove the current restrictions, allowing initialized constants, generic in parameters, and aggregates of limited type. This should actually simplify compilers, rather than make them more complicated.

For additional references to Section 7. of ANSI/MIL-STD-1815A, see the following sections, revision request numbers, and revision request titles in this document.

REVISION REQUEST

<u>NUMBER</u>	<u>TITLE</u>	<u>SECTION</u>
0117	PRE-ELABORATION	3
0125	INTRODUCE INHERITANCE IN ADA	3
0092	FINALIZATION	3
0094	IDENTIFIER LISTS AND THE EQUIVALENCE OF SINGLE AND MULTIPLE DECLARATIONS	3
0096	LIMITATIONS ON USE OF RENAMING	8
0101	IMPLEMENTATION OF EXCEPTIONS AS TYPES	11

**ADA 9X REVISION REQUESTS
THAT REFERENCE
ANSI/MIL-STD-1815A**

SECTION 8. VISIBILITY RULES

VISIBILITY OF OPERATORS BETWEEN PACKAGES**DATE:** March 7, 1989**NAME:** B C Tooby (from material provided by several organizations working in Ada applications, including High Integrity Systems)**ADDRESS:** High Integrity Systems
Astra Centre
Edinburgh Way
Harlow, Essex CM20 2BE
UK**TELEPHONE:** +44 279 450000**ANSI/MIL-STD-1815A REFERENCE:** 8.**PROBLEM:**

In large Ada programs, it is good practice to use expanded names when making references between packages, and to restrict the use of the use clause. Not doing so makes it very difficult to understand and maintain a large program, since the reader is not sure which names are locally defined, and which names are defined somewhere else in a huge stack of source listings.

In practice a great deal of code is written which makes reference to types declared in other packages - as is natural to Ada - but operators such as "+" which are associated with those types are not visible. One can not therefore write, for example

```
with Calendar;  
procedure Example is  
  Interval: Duration;  
  Previous_Time : Calendar.Time := Calendar.Clock'  
begin
```

```
  ...  
  Internal := Calendar.Clock - Previous_Time  
end Example;
```

This simple example under-states how widespread the occurrence of the problem is in a typical real system. Often a significant proportion of packages have to make references to types, and to implicitly or explicitly defined operators, declared in other packages.

What actually happens is that programmers use clauses to make the operators visible, and then stop using expanded names (often inadvertently) since the compiler can no longer check that they are doing so.

Alternative solutions to the use clause are seen as cumbersome or ugly.

IMPORTANCE: IMPORTANT

Rigorous enforcement of a Code of Practice can prevent the effects of the problem; but it would be nice if the language provided a way of allowing the compiler to assist.

CURRENT WORKAROUNDS:

The most common workaround we have seen is to allow the use of a locally-defined use clause, with a comment such as

```
-- only to make operators visible
```

the intention being that expanded names should still be used for everything other than operators.

By far less common is to declare local functions which 'rename' the wanted operator, or to call the operators as ordinary functions. Having to do so is seen as a pain in the neck, and something which adds unnecessary clutter or ugliness to the finished program.

POSSIBLE SOLUTIONS:

A solution is sought which enables operators such as "+" to be made visible without having to make everything else in the corresponding specification visible.

LIMITATIONS ON USE OF RENAMING

DATE: March 22, 1989

NAME: E.N. Thomas

DISCLAIMER: The views expressed in this note are those of the author, and do not necessarily represent those of SD-Scicon PLC.

ADDRESS: SD-Scicon PLC
Pembroke House
Pembroke Broadway
CAMBERLEY
Surrey
UK
GU15 3XD

TELEPHONE: +44 276 686200

ANSI/MIL-STD-1815A REFERENCE: 8.5, 3.5.1, 3.8.1, 3.9, 6.3, 7.4.1

PROBLEM:

Renaming has some unnecessary limitations, some of which seem to have been historical accidents in the design of the language, and some oversights.

Section 8.5(9) allows enumeration literals to be renamed as functions. Unfortunately, the form of renaming declarations given in 8.5(2) does not permit the same syntax for the new name as that for enumeration literals in section 3.5.1(2). In particular, it is not possible to introduce for an enumeration literal a new name that is a character literal, although an existing character literal can be renamed using an identifier.

IMPORTANCE: IMPORTANT

Regularity is compromised without these, and the design goals of 1.3(3) are violated ("a small number of underlying concepts integrated in a consistent and systematic way").

CURRENT WORKAROUNDS:

There is no workaround for the case of renaming as a character literal.

Providing a procedure body where renames would suffice requires writing a body that calls the relevant one. This has the disadvantage that it relies on efficient treatment of inline substitution if run-time penalties are to be avoided.

Providing a full type where renames would suffice requires use of a derived type, or of a record type with a single component of the required type. Both of these have disadvantages in the body where the equivalence of the types is known, in that explicit conversions of component selection need to be used.

POSSIBLE SOLUTIONS:

Permit renames to be used for the purposes of supplying subprogram bodies, and full type declarations.

In the case of full type declarations, it would be clearer if the syntax of 8.5 was extended to allow the form below. Requirements for conformance of discriminant parts in 3.8.1(3) and 7.4.1(3) would need to apply for such renamings too, and the syntax for renames seems better off without allowed [discriminant_part].

```
renaming_declaration ::=  
    <<as existing, followed by>>  
    | type identifier renames type_name;
```

VISIBILITY CONTROL

DATE: March 23, 1989

NAME: John Dawes (from material prepared by members of Ada UK)

ADDRESS: ICL
Eskdale Road
Winnersh
Wokingham
Berkshire RG11 5TT
UK

TELEPHONE: +44 734 693131

ANSI/MIL-STD-1815A REFERENCE: 8.5

PROBLEM:

The problem typically arises when a library unit imports type declarations from elsewhere, provides some additional functionality, e.g., by visible subprograms for the types, and wishes to make that functionality available to other library units without those units having to "with" the units whence the types originally came. This requirement typically arises from abstract-data-type and object-oriented design methods.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

At present type renaming is supported only by the use of subtype or derived type declarations. The former requires renaming of enumeration literals as functions, and the latter requires the use of explicit type conversion all over the place. Neither provides an adequate solution for large programs.

POSSIBLE SOLUTIONS:

Provide a type renaming construct which allows a unit to give a new name to an imported type; the visibility semantics of the to be the same as for derived type declaration. It is believed that this has no major ramifications and is essentially a cosmetic change.

The following example illustrates the problem, the current workarounds, and the proposed solution.

```
package A is                -- original exporting package
  type T is (X,Y);
  procedure F(P:T);
end F;
end A;

with A;                      -- "subclass" package
package B1 is

  type T1 is new A.T;        -- first workaround
  procedure F(P:T1) and enumeration literals X and Y
```

```

--      are implicitly declared

      subtype T2 is A.T;      -- second workaround
--      no implicit declarations; it is necessary to redeclare
--      anything needed as e.g.:
          function X return A.T renames A.X;

      type T3 renames A.T; -- proposed solution
--      F, X, and Y are implicitly redeclared

--      subprograms for types T1, T2, and T3;
end B;

with A;
package B2 is                -- another "subclass" package
    type T1 is new A.T;
    subtype T2 is A.T;
        function X return A.T renames A.X;
    type T3 renames A.T;
--      more subprograms for types T1, T2, and T3;
end B;

with B1, B2;                -- end user package
package C is
    Obj11: B1.T1; -- all operations available
    Obj12: B1.T2; -- basic operations only
    Obj13: B1.T3; -- all operations available

    Obj21: B2.T1; -- different type to Obj11
    Obj22: B2.T2; -- same type as Obj12
    Obj23: B2.T3; -- same type as Obj13
--
--      Note that use of the derived types B1.T1 and B2.T1 is
--      difficult because they are different types and explicit type
--      conversions are required. Use of the subtypes B1.T2 and
--      B2.T2 is difficult as the nonbasic operations are not
--      visible (unless renamed as for enumeration literal X). The
--      proposed renamed types B1.T3 and B2.T3 suffer from neither
--      of these disadvantages.
--
...
end C;

```

EXTENDED CHARACTER SET

DATE: March 9, 1989
NAME: SY Wong
ADDRESS: 5200 Topeka Drive
Tarzana, CA 91356
TELEPHONE: 818-345-6274
E-mail: hermix!sywong@rand-unix.arpa

ANSI/MIL-STD-1815a REFERENCE: 8.6

PROBLEM:

Extended character set place holder needed NOW to preserve portability of code for 9X.

Requirements of extended character set are:

1. Allow character'pos(character'last) agree with the universal representation of 8-bits.
2. Preserve present ease of representing ASCII graphical characters with ASCII graphical symbols, i.e., compatibility with present 7-bit ASCII definition.
3. Allow application/country defined extended part of character code and multi-character representation of Asian characters, NOT 16-bit integers. Two 7-bit characters do not provide enough coding. There is no need for Ada to define Asian characters.
4. Prevent proliferation of privately defined string types of special character definitions.
5. Permit easy definition of multi-character coding of Asian characters and yet using a single Ada string definition and operations.
6. Allow intermixed English and Asian characters without having to require all English characters be represented by multi-character codings.
7. Allow legal implementation earlier than 9X to save millions of lines of code from conversion.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Proliferation of individual definitions and string handling packages.

POSSIBLE SOLUTIONS:

Redefine:

type standard.character is (nul, .. del, c128, .. c255);
for standard.character use (0, .. 255);

-- DO NOT FIX THE LAST 128 CHARACTERS TO ANY SYMBOL SET.
-- This avoids interminable argument of whose symbol set.

No change in package ASCII.

No change in string definition, except possibly replacing
positive by natural in index range, to allow easy conversion
to/from array (character) of character.

Application/country can define symbol binding outside of
package standard, for example:

package fonechar is -- for any part or all of the character set:

```
--  
aah: constant character:= c137;  
eeh: constant character:= c138;  
ooh: constant character:= c139;  
-- special display may print aah,eeh and ooh symbols.  
next_sequence_chinese: constant character:= c255;  
end fonechar;
```

There is no need to define the chinese characters in Ada style. Since character comparison is on positional value basis, multi-character coding does not affect string comparisons. No special Asian string operation is required. An exception handler may be required if string character count is not even.

Pre-9X legalization of 8-bit character will facilitate many Ada binding efforts now in progress. The incompatibility of C and Ada characters is a serious problem for UNIX interface definition and portability.

For additional references to Section 8. of ANSI/MIL-STD-1815A, see the following sections, revision request numbers, and revision request titles in this document.

REVISION REQUEST

<u>NUMBER</u>	<u>TITLE</u>	<u>SECTION</u>
0019	VARIABLE FINALIZATION	3
0022	VISIBILITY OF BASIC OPERATIONS ON A TYPE	3
0101	IMPLEMENTATION OF EXCEPTIONS AS TYPES	11

**ADA 9X REVISION REQUESTS
THAT REFERENCE
ANSI/MIL-STD-1815A**

SECTION 9. TASKS

TASK PRIORITIES AND ENTRY FAMILIES**DATE:** March 7, 1989**NAME:** B C Tooby**ADDRESS:** High Integrity Systems
Astra Centre
Edinburgh Way
Harlow, Essex CM20 2BE
UK**TELEPHONE:** +44 279 450000**ANSI/MIL-STD-1815A REFERENCE:** 9.**PROBLEM:**

This is actually a request that a proposed revision not be made.

Suggestions for improving the Ada task priority system have included a suggestion that Entry Families should be removed (since they would then no longer be necessary).

I wish to point out that Entry Families serve at least one existing design purpose other than the one of providing prioritized entry queues:

In a hierarchical modular design, it is a useful principle that references be made downwards in the hierarchy (towards more junior levels), or within the same level, but not upwards. Seniority in the hierarchy might be determined, for example, by the level of abstraction of the related data.

It can happen that a high level, active module consists of a number of task objects, each of the same type. For example, each task object deals with a particular Communications Call (not to be confused with entry call) at the most abstract level. Such a collection of high level objects may need to make entry calls downwards in the hierarchy in order to acquire data from a lower-level module. Such a lower-level module may determine that information is arriving which is relevant to a particular Communications Call at any moment in time, and may wish to accept and entry call from the appropriate high-level task object, in order to pass data upwards (via an out parameter).

In this situation an Entry Family in the lower-level module (one member per concurrent Communications Call) allows the desired hierarchical principles to be preserved.

While other design solutions can be devised, the removal of Entry Families would prevent what turns out to be a simple and well-structured design in this type of situation.

IMPORTANCE:

Perhaps **ESSENTIAL**: in the sense that if Entry Families are removed in the revision, the revised standard is unlikely to be accepted by organizations with existing designs which use Entry Families (at least for non-prioritizing purposes).

CURRENT WORKAROUNDS:

POSSIBLE SOLUTIONS:

REDUCING RUN-TIME TASKING OVERHEAD**DATE:** April 19, 1989**NAME:** Daniel L. Stock

Ted Baker

ADDRESS: R.R. Software, Inc.
2145 Crooks Road #50
Troy, MI 48084-3183Department of Computer Science
Florida State University
Tallahassee, FL 32306-4019**TELEPHONE:**

(313) 643-6370

(904) 644-5452

E-mail: stockd@ajpo.sei.cmu.edu

E-mail: tbaker@ajpo.sei.cmu.edu

E-mail: baker@nu.cs.fsu.edu

ANSI/MIL-STD-1815A REFERENCE: 9.**PROBLEM:**

Ada tasking involves too much run-time overhead for some high-performance applications, including many embedded systems applications for which the language was designed. This overhead not only slows down the program in general, but may also occur at unpredictable times, thus delaying response at critical times. To avoid the overhead, real-time programmers frequently circumvent Ada tasking.

The problem is exacerbated by Ada's lack of support for those who do try to use tasking in an efficient manner. There is no standard set of guidelines to programmers for writing optimizable tasking code, or to language implementors, for deciding which optimizations to perform. Also, there is no simple way for a programmer who is concerned with writing portable high-performance code to check that optimizations applied under one implementation will be applied under different implementations.

The consequences of Ada tasking overhead have not gone unnoticed in higher circles of government. A recent General Accounting Office report [1] noted that Ada has limitations in real-time applications that require fast processing speed, compact computer programs, and accurate timing control. All three of these requirements are directly and adversely affected by Ada's current tasking overhead.

A complete solution to the problem appears unlikely: many features of the Ada tasking model require run-time overhead. A partial solution should allow reducing the overhead in many cases, while retaining as much of the current tasking model as possible in those cases. Both language implementers and application programmers should have guidelines to facilitate portable efficient use of tasking.

IMPORTANCE: IMPORTANT**CURRENT WORKAROUNDS:**

Ada's tasking overhead has affected many users of the language. One common workaround is to rewrite parts of the underlying Ada run-time system, customizing it to the application; another is to interface to a different language (usually assembly language) for tasking purposes. Both of these approaches sacrifice much of the portability, reliability, and maintainability for which Ada was designed.

A third workaround is commonly suggested: get faster hardware. This is often impractical, or contractually

impossible. Furthermore, the ability to use hardware to its greatest potential is always a virtue in a language. Ada generally does a good job of providing this ability for sequential programming, so why not also for concurrent programming?

A final workaround is to use only some of Ada's tasking features. This avoids most of the difficulties of the other workarounds; however, there is currently no reason for a programmer to believe that this approach will be effective, especially if the application (or parts thereof) is to be ported to a different run-time system. Thus, the main problem with this approach is that Ada provides no standardized guidance as to which tasking features to avoid.

POSSIBLE SOLUTION:

By having a uniform convention (or perhaps several) to follow when choosing which Ada tasking features to use, the last workaround could be made practical. A programmer who follows this convention should have a reasonable expectation that the permitted constructs will execute substantially faster than they would in the general case, on most implementations.

Baker has done an experiment, in which run-time code for rendezvous was simplified, eliminating all the code and data structures associated with dynamic tasking, abortion, and time-outs. The times for general call-accept rendezvous were reduced by about 25%. This experiment indicates some significant potential savings.

A better reason for encouraging programmers and implementors to agree on a convention for a restricted model of tasking is made by recent work at Carnegie Mellon University [2,3,4] on the application of priority-inheritance protocols to Ada client-server systems. This shows that sticking to a restricted tasking model can permit much a higher level of performance and assurance of meeting timing constraints.

The CMU work builds on earlier work involving reduction of "passive" tasks to traditional monitors, protected by semaphores. These optimizations have proven very successful. Furthermore, they can help avoid two other dangers of Ada tasking: priority inversion and deadlock. The main difficulty is that not all compilers support these optimizations, and there is no standardized set of rules for a programmer to follow when writing tasks, to insure that a compiler will be able to perform this kind of optimization. (Ada 9X may also need to slightly relax the current Ada priority rules to allow these optimizations. For example, a task should be permitted to execute at a higher effective priority when a higher-priority task is waiting on one of the first task's entry queues; such changes are desirable in any case to reduce priority inversion.)

A third good reason for recognizing restricted tasking is the demand for some form of fast "abort and restart" capability. It appears impossible to provide this in a form that is completely general and safe, while still meeting the performance demands of time-critical systems. However, if an Ada implementation were certain that a particular program used only a restricted form of tasking, it might be possible to take short-cuts that could lead to the kind of performance that is required.

Note that we are not proposing to throw out Ada tasking, or to allow subset implementations. An Ada language implementation would still be required to support the full tasking model for programs that use it.

What we would like to see is a STANDARD programming convention (or several such conventions), under which it is reasonable to expect high-performance tasking. This would give programmers and compiler-builders a common, well-defined target. It would also give compiler buyers something specific to look for when shopping for an Ada compiler.

We would also like to see the language provide a way for a programmer to specify that the compiler should check that the convention has been followed, if the user so desires. For example, the language could define a pragma `FAST_TASKING`. This pragma, like several of the current pragmas, might be permitted only at the start of the first compilation when creating a new program library. After this pragma has been compiled, a fixed set of Ada tasking features would be considered illegal. (Note that such a pragma would have to be defined by the language, since implementation-defined pragmas may not affect the legality of a compilation.) Exactly which features would be disallowed is a topic for discussion, but the idea would be to allow a simpler, faster run-time environment to be used.

This solution initially imposes little burden on implementors, since an existing run-time system could be used even if the convention is followed. However, the marketplace is likely to induce them quickly to do the right thing: provide a faster run-time environment and/or better code generation when one does follow the convention.

REFERENCES:

- [1] General Accounting Office, "Status, Costs, and Issues Associated With Defense's Implementation of Ada", report commissioned by the House Appropriations Subcommittee on Defense (1989).
- [2] J. Goodenough and L. Sha, "The Priority Ceiling Protocol: A Method for Minimizing the Blocking of High Priority Ada Tasks", Proceedings of the Second International Workshop on Real-Time Ada Issues, *Ada Letters* Volume VIII, Number 7 (Fall 1988), pp. 20-31.
- [3] C.D. Locke and J. Goodenough, "A Practical Application of the Ceiling Protocol in a Real-Time System", Proceedings of the Second International Workshop on Real-Time Ada Issues, *Ada Letters* Volume VIII, Number 7 (Fall 1988), pp. 35-38.
- [4] L. Sha, R. Rajkumar, and J.P. Lehoczky, "Priority Inheritance Protocols, An Approach to Real-Time Synchronization", technical report CMU-CS-87-181, Carnegie Mellon University (November 1987).

ATTRIBUTES FOR TASK ARRAY COMPONENTS**DATE:** May 23, 1989**NAME:** Jeffrey R. Carter**ADDRESS:** Martin Marietta Astronautics Group
MS L0330
P.O. Box 179
Denver, CO 80201**TELEPHONE:** (303) 971-4850
(303) 971-6817**ANSI/MIL-STD-1815A REFERENCE:** 9.**PROBLEM:**

Task objects which are components of an array must be explicitly told their indices if they need to know them. In addition, a master with multiple dependent tasks can only determine if all its dependent tasks have terminated by individually checking each dependent task. This can have a serious impact on the use of Ada on massively parallel systems.

For example, a computer with 2^{20} processors can perform the multiplication of two 1,000 X 1,000 matrices as fast as two 2 X 2 matrices. In Ada, this could be represented as an array with components of some task type which will compute the value of one component of the result. The master multiplication function must step through the array and tell each task which component to compute. This puts a significant serial overhead on this inherently parallel function.

In addition, the master must check each task for termination before executing the return statement, as the evaluation of the return expression may give erroneous results if it is done before all of the tasks have terminated. This puts another significant serial overhead on the function.

IMPORTANCE: VARIABLE

May be **ESSENTIAL** to those using Ada on massively parallel systems.

CURRENT WORKAROUNDS: NONE**POSSIBLE SOLUTIONS:**

Add the following attributes to the standard:

```
task_name'index_type(n); -- n is any integer type--returns the
    type of the nth index of the array of which the task is a component--raises
    program_error if the task is not a component of an array--raises constraint_error
    if n <= 0 or if n > number of dimensions of the array
task_name'index_value (n); -- n is any integer type--returns
    the value of the nth index of the array of which the task is a component--raises
    program_error if the task is not a component of an array--raises constraint_error
    if n <= 0 or if N > number of dimensions of the array
```

These are similar to the 'last, 'first, and 'length attributes for arrays. If used out a value of n, return the value for n = 1.

```
unit_name'all_terminated;--returns false if the unit has an
                           un_terminated dependent task--returns true otherwise
```

Thus a task type could contain statements such as

```
task body multier is
  row : multier'index_type := multier'index_value;
while its master could contain
  wait_for_termination : while not ""all_terminated loop
    null;
  end loop wait_for_termination;
return result;
```

These attributes would be compatible with the current standard.

ASYNCHRONOUS EVENT HANDLING

DATE: June 7, 1989

NAME: Ted Baker (ACM Special Interest Group on Ada, Ada Runtime Environment Working Group)

ADDRESS: Department of Computer Science
Florida State University
Tallahassee, FL 32306-4019

TELEPHONE: (904) 644-5452
E-mail: (ARPAnet) tbaker@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 9.

PROBLEM:

Certain kinds of real-time events require that one or more tasks respond immediately, by following an alternate control path. We will call this an asynchronous transfer of control (AST). There does not appear to be a sufficiently efficient way to program solutions to these events in the present Ada language.

SPECIFIC REQUIREMENTS/SOLUTION CRITERIA:

In brief, a solution should:

- 1) stop a task immediately (ideally, preemptively);
- 2) force a task to resume on a different control path;
- 3) resume right away, or later, under programmer control;
- 4) be very fast, so that it does not add to processor load;
- 5) be repeatable without bound.

IMPORTANCE:

The lack of a standard solution to this problem within the Ada language forces users of highly critical and fault-tolerant systems choose between:

- a) not using Ada;
- b) unsafe, nonportable "hackery" (i.e. local solutions based on a particular machine, compiler, and runtime environment; such as using machine-code to reset program counter and stack pointer register);
- c) producing an inferior end product, that cannot respond to asynchronous events in a timely fashion.

CURRENT WORKAROUNDS:

A prime example of the kind of event that may require an AST is a timing fault, where a task fails to complete a time-critical computation on time. Two important recovery strategies for such a fault are: (1) for the offending task to abandon its sequential execution and apply a short-cut algorithm; (2) for the task to stop executing immediately and restart with fresh data the next time it is scheduled to execute.

Other important examples of situations requiring an AST include failure of a hardware component (e.g. a processor that may be executing another task, in a distributed environment), and a change of operational mode (e.g. aborting a bombing run to engage in a defensive maneuver).

The kind of events that are candidates for an AST cannot be handled by means of the standard interrupt-entry binding mechanism, because the desired response to the event is to force one or more tasks to immediately abandon their sequential execution -- preemptively and asynchronously. That is, the application cannot wait until such a task next reaches an ACCEPT statement for an entry, and can not always separate the response associated with the event into an independent task that is always ready to accept an entry call.

Events requiring an AST arise under conditions where they must be handled quickly, within strict time constraints. For example, timing failures often are caused by transient processor overloads. During such a period of overload it would be disastrous if the recovery mechanism (e.g. aborting a task and creating a new one) adds to the processor load.

The solution must permit the processor load imposed by certain tasks to be removed immediately, while allowing other tasks to respond to the changed situation in a timely fashion.

The solution must permit the computations of certain tasks to be cut off immediately, so that they cannot proceed along their current thread of control. Suspension is not enough; there must be a mechanism to insure that when a task resumes execution it will resume at a known point. (For example, if a bombing run is aborted, the tasks responsible should restart from the beginning, with fresh data, the next time a bombing run is ordered.) In some cases the resumption of execution must be immediate.

The solution must be implementable in a way that supports an unbounded number of asynchronous events to be handled, without recourse to full system restart. For example, a solution that consumes (permanently) one byte of memory each time such a fault occurs would not be acceptable.

POSSIBLE SOLUTIONS:

This problem has been raised by many developers of avionics systems, as well as developers of other hard real-time applications. It has also been raised by people attempting to program fault-tolerant multiprocessor systems. It has been among the top problems discussed at the Real-Time Ada Workshops held in Devon, England, the past two years [1,2,3].

The following approaches to solutions and workarounds have been proposed:

- 1) the use of the existing exception mechanism, with exceptions being raised preemptively and asynchronously, via the runtime environment;
- 2) the use of task abortion (and re-creation);
- 3) the use of continual polling by all tasks that may need to respond to asynchronous events;
- 4) introduction of one or more new language constructs, such as a task restart operation and abortable procedure calls;
- 5) ad hoc implementation-specific solutions, by recourse to machine-code.

DIFFICULTIES TO BE CONSIDERED:

Support of immediate asynchronous transfer of control in an efficient manner will require careful design of the entire runtime model. Most existing compilers have not been designed with this in mind.

REFERENCES:

- [1] P. de Bondeli, "Asynchronous Transfer of Control" (Session Summary), Proceedings of the International Workshop on Real-Time Ada Issues, Ada Letters VIII,7 (Fall 1988).
- [2] T. P. Baker, "Improving Immediacy in Ada", Proceedings of the International Workshop on Real-Time Ada Issues, Ada Letters VIII,7 (Fall 1988).
- [3] W.J. Toetenel and J. Van Katwijk, "Asynchronous Transfer of Control in Ada", Proceedings of the International Workshop on Real-Time Ada Issues, Ada Letters VIII,7 (Fall 1988).

DISTRIBUTED SYSTEMS**DATE:** June 7, 1989**NAME:** A.J. Wellings (A CM Special Interest Group on Ada, Ada Runtime Environment Working Group)**ADDRESS:** Department of Computer Science
University of York
Heslington
York
YO1 5DD
UK**TELEPHONE:****ANSI/MIL-STD-1815A REFERENCE:** 9., 13.**PROBLEM:**

Computer architectures vary widely, ranging from single processor systems to multicomputers without shared memory connected by networks. Many embedded systems applications must be distributed across these architectures; it is therefore a requirement that it should be possible to program such systems in Ada. Although there are many ways in which Ada can be used to program distributed applications, two extremes can be identified: the multiple Ada program approach and the single Ada program approach. The language should not hinder or prevent distributed programs being written using either approach (or both).

Currently, ambiguity surrounds some Ada language features, making it very difficult to distribute single Ada programs. These ambiguities include: the definition of timed and conditional entry calls, the effect of hardware failures, and the definition of packages SYSTEM and STANDARD.

SPECIFIC REQUIREMENT/SOLUTION CRITERIA:

Ada should contain no features that prohibit the distribution of a single Ada program across homogeneous or heterogeneous processors.

IMPORTANCE:

Designers will resort to operating system support for distributed systems or will use non-portable interpretations. Ada tasking may not be used.

CURRENT WORKAROUNDS:

The Ada language does not impose any limitations, or demand any semantics, associated with multiple communicating Ada programs. Consequently, implementors are free to make their own provisions. However, if a single Ada program is to be distributed, the full language semantics must be supported. It should be possible to distribute a single Ada program across homogeneous or heterogeneous computer architectures.

The problem of writing distributed systems in Ada has become very topical over the last few years. Many different approaches have emerged [1,3,4,5,6,8]. For a review, see one of the following

references [2,3,7].

It is not surprising that, given all these different projects, the Ada community has failed to agree on a preferred approach to distribution; as was very apparent at the last two Real-time Ada Workshops[10,11]. However, it is possible to get some agreement on those features of Ada that hinder or prevent distributed programs being written in Ada.

To alleviate the current situation, there are two possible strategies that Ada 9x can adopt. The first is to define a preferred distribution approach, including: the units of partitioning; the means of inter-unit communication and its precise definition; and the semantics of the program in the presence of processor and communication failure. This strategy is the one which has recently been adopted by languages whose primary application domain is distributed systems (e.g. Argus, SR and CONIC). If adopted by Ada, it could have a major impact on the language definition.

The alternative strategy is to not prescribe a preferred approach but merely to address those language issues which make any approach problematic. The current minimum change/maximum impact philosophy to Ada 9x would suggest that the latter strategy is the one that should be supported. However, the result may well be that it will remain very difficult to write transportable distributed software in Ada. We shall therefore concentrate on those issues which inhibit the distribution of a single Ada program.

1) Timed and Conditional Entry Calls

It has been known for some time that the current interpretation of the semantics of timed and conditional entry calls causes two problems in the context of distributed execution. The first is the statement that a timed entry call with a specified duration of 0 (or a negative value) is equivalent to a conditional entry call. One valid interpretation causes anomalous behavior for timed entry calls where the timeout value is approximately equal to network delays (d) associated with the underlying implementation protocol. It is possible for a timed entry call with a delay "d" to succeed, a call with delay "d - very small value" to fail, and a delay 0 to succeed.

The second problem concerns the use of timed entry calls as a means of protecting a task against long network delays, network partitioning and processor failure. This clearly falls in the domain of fault tolerance so we will not discuss the problem in detail.

For a full discussion on both of the above issues see Volz and Mudge[9].

2) Package System

Package System, defined in Chapter 13 of the Ada Reference Manual, assumes that programs will be implemented on single processor systems. For homogeneous systems this causes difficulty with MEMORY_SIZE. Note that there is nothing to stop the ADDRESS type being a record structure which contains a machine address as well as an address within the machine. However, MEMORY_SIZE must be of type universal_integer. One is therefore restricted to a single constant for the whole system; what is needed is a way of representing the available memory at each node in the distributed architecture.

With heterogeneous systems there is the additional problem of the constants that define the representation of integer etc. A single set of these constants is clearly not adequate.

There are similar difficulties with package STANDARD.

POSSIBLE SOLUTIONS:

Timed and conditional entry call should be redefined to specify their exact meaning in a distributed environment. Packages SYSTEM and STANDARD must be changed so that local (and possibly dynamic) views can be incorporated.

DIFFICULTIES TO BE CONSIDERED:

The main difficulty appears to be in deciding exactly how far the Ada language should go in its support for distributed programs; in particular to what extent the language should define a program's operation in the presence of processor and network failures.

For heterogeneous systems there may be problems that a language revision cannot address. For example differing data representations for logically identical constants.

REFERENCES/SUPPORTING MATERIAL:

- [1] C. Atkinson, T. Moreton and A. Natali, Ada for Distributed Systems, Ada Companion Series, Cambridge University Press (1988).
- [2] A. Burns, A.M. Lister and A.J. Wellings, A Review of Ada Tasking, Lecture Notes in Computer Science, Volume 262, Springer-Verlag (1987).
- [3] D. Cornhill, "A Survivable Distributed Computing System for Embedded Application Programs Written in Ada", Ada Letters (November/December 1983).
- [4] A.D. Hutcheon and A.J. Wellings, "The Virtual Node Approach to Designing Distributed Ada Programs", Ada User, Vol. 9(Supplement), pp. 35-42 (December 1988).
- [5] R. Jha, G. Eisenhauer, J.M. Kamrad II and D. Cornhill, "An Implementation Supporting Distributed Execution of Partitioned Ada Programs", Ada Letters, Vol. 9(1) (January/February 1989).
- [6] D. Keefe, G. M. Tomlinson, I. C. Wand and A. J. Wellings, PULSE: An Ada-based Distributed Operating System, APIC Studies in Data Processing Series, Academic Press (1985).
- [7] M. Tedd, S. Crespi-Reghizzi and A. Natali, Ada for Multi-microprocessors, The Ada Companion Series, Cambridge University Press (1984).
- [8] M. Volz, T. Mudge, A.W. Naylor and J.H. Mayer, "Some Problems in Distributing Real-Time Ada Programs Across Machines", pp. 72-84 in Ada in Use, The Ada Companion Series, ed. J.G.P. Barnes and G.A. Fisher, Cambridge University Press (1985).
- [9] R.A. Volz and T.N. Mudge, "Timing Issues in the Distributed Execution of Ada Programs", IEEE Transactions on Computers, Vol. C-36(4), pp. 449-459 (April 1987).
- [10] A.J. Wellings, "Issues in Distributed Processing - Session Summary", Proceedings of the 1st International Workshop on Real-Time Ada Issues, ACM Ada Letters, Vol. 7(6), pp. 57-60 (October 1987).
- [11] A.J. Wellings, "Distributed Execution: Units of Partitioning - Session Summary", Proceedings of the 2nd International Workshop on Real-Time Ada Issues, ACM Ada Letters, Ada Letters, Vol. 8(7), pp. 85-88 (1988).

DISCRIMINATE VALUES PASSED AT TASK OBJECT CREATION

DATE: May 30, 1989

NAME: Donald R. Clarson

ADDRESS: Teledyne Brown Engineering
Industrial Way East
Eatontown, NJ 07724

TELEPHONE: (201) 389-7500

ANSI/MIL-STD-1815A Reference: 9.1(3)

PROBLEM:

The task body cannot reference an identifier which is unique to the task object currently executing unless the identifying value is passed during rendezvous after task activation begins.

IMPORTANCE: IMPORTANT

Allocation of resources to each task object requires the overhead of task rendezvous to pass the identify the task in execution.

CURRENT WORKAROUNDS:

Several schemes have been proposed to distribute the overhead of the required task rendezvous but these may delay the identification until the activation of the task is completed [2]. None of these eliminates the overhead of task rendezvous.

POSSIBLE SOLUTIONS:

Abstract: This paper proposes an addition to the Ada Programming Language which will allow a set of values to be passed as discriminant values when a task object is created. This feature will be useful to avoid an additional task entry call following task activation for some applications. Syntax and semantic rules are presented together with several contexts where the feature may be used.

Introduction: Some applications for Ada tasks require each task to have visibility to a unique set of values which identify the particular values and variables to be used by the task. This allows a single task type to be declared with each of several tasks allotted a portion of the processing required.

One method for passing these values is for the master task to call an additional entry of each task following the activation of the set of tasks. The values passed are stored in a variable in the task body and used throughout the execution of the statements of the task. This requires additional statements in both the master task and the body of the dependent tasks and requires additional time for this call to be completed before each task begins execution of the algorithm.

Other more efficient initialization routines for multiprocessor systems presented by A. Burns in Ada Letters Vol. V, pp. 1-55 .. 60[1] distribute these additional rendezvous over the available multiprocessors. The requirement for these additional entry calls could be eliminated if these initial values were associated with the task when the task was created. The scope of these values extends to the end of the task body. Since

the task being created has not yet begun execution these values could be stored as part of the task object or could be asynchronously posted to be passed to the task at the start of task activation by the run-time environment.

A language feature is needed to allow the programmer to specify these values when the task object is created. This additional feature should be an optional addition to an existing language construct so that existing Ada programs will still be correct.

Ada tasks have characteristics of program units which are invoked by activation and characteristics of constant objects which store values of the task type.

One possible approach to passing the initial set of values to each task would be to allow the task specification to have an optional formal part with a number of parameter specifications restricted to mode in for the values to be passed during task activation. This would require additional syntactic forms since the activation of a task is invoked implicitly. The association of actual parameters would have to be added to the semantics of task creation.

A simpler approach is to allow an optional discriminant part for a task type specification. Task types allow task objects to be declared while keeping the structure of the object visible only to the run-time environment. Task types are limited so assignment and comparison for equality operations are not predefined.

The semantics of the existing language constructs allow a discriminant part in the specification for limited private types and allow values to be associated with these discriminant components when objects are created. These discriminants are visible as components of the object but may not be assigned new values since assignment of a complete record value is not allowed for a limited type.

Proposal: The proposed syntax for a task specification is:

```
task_specification ::=
  task [type] identifier [discriminant_part] [ is
    {entry_declaration}
    {representation_clause}
  end [task_simple_name] ]
```

The semantics allow the discriminant part only for a task type specification and not for a task specification which defines a single task. This is consistent with existing constructs which allow a discriminant part for type declarations and not the declaration of objects of an anonymous type.

The rules for visibility of the discriminant components follow existing rules for discriminant components for objects of a limited private type and for task entries. The discriminant component is visible by component selection with the task object name as the prefix. Within the (specification and) body of the task the discriminant values for the currently executing task are visible using the simple discriminant name or an extended name with the task unit name as prefix.

Discussion: Since this optional feature uses existing language constructs, the implementation should be straightforward. The task object requires additional discrete components for the discriminant values. The elaboration of the task object requires the evaluation of the discriminant constraint. The discriminant values may be passed as part of the activation record when the created task does not share memory with the master task.

This additional processing represents actions which would otherwise occur during the additional entry call following task activation. This additional language feature should be considerably more efficient than using

existing language constructs to pass these values since the additional synchronization is not needed. The TASK_NAME_SERVER outlined by Cheng [2] could be used by the creating task if the Internal_ID_of_Task were discrete. A function call to a function with side-effects would be sufficient if the creating task were the only caller since the creating task is synchronized with the created task prior to the activation of the task.

A representation clause for the discriminant components is not allowed since the representation of the task object is known to the run-time environment and is not determined in the declarative part which declares the task type.

Examples: The following are some of the contexts where a discriminant constraint may be useful. The expressions provided for the discriminant constraint may safely use functions with side effects since these are evaluated by a single thread of control when the task is created.

Single task object declaration:

```
T1 : Task_type_name [discriminant_constraint];
      --elaboration evaluates the expressions
      -- of the discriminant constraint
```

Record type with task component

type RT is record

```
T1 : Task_type_name [discriminant_constraint] ;
      --elaboration of the object declaration
      --evaluates the expressions
      --of the discriminant constraint
```

end record;

```
T2 : RT ;          -- task object declaration
```

The example of task initialization presented by Burns[1] becomes:

```
procedure MAIN is
  MAX : constant := 1023;
  subtype TASK_RANGE is Integer range 0 .. MAX;
  task type T (MY_NUMBER : TASK_RANGE ); --discriminant part
  type T_PTR is access T;
  SET : array ( TASK_RANGE )
        of T_PTR ;
  task body T is
    -- ... other declarations
  begin
    -- ... statements for algorithms
  end T;
begin
  for I in TASK_RANGE loop
    SET (I) := new T (I); --each task is created and
  end loop;              --initialized in turn
end MAIN;                --with no rendezvous necessary
```

Summary: The addition of an optional discriminant part for task type specifications fulfills a requirement

for some tasking applications which now use less efficient language features. This addition is compatible with existing Ada programs and can be implemented in a straightforward manner.

REFERENCES:

- [1] Burns, A. Efficient Initialization Routines For Microprocessor Systems Programmed in Ada, Ada Letters, Vol. V, No.1, pp. 55-60, July/August 1985.
- [2] Cheng, J. and Ushijima, K., Naming Ada Tasks at Run-Time, Ada Letters, Vol. IX, No. 2, pp. 52-61, March/April 1989.

MUTATION OF TYPES

DATE: November 23, 1988

NAME: H T (Ted) Goranson
(Formerly with SIRIUS, Inc.)

ADDRESS: Science Applications International Corp.
Mail Stop T3-5
1710 Goodridge Drive
McLean, VA 22102

TELEPHONE: Office: 703/556-7118
Work: 804/426-6704

ANSI/MIL-STD-1815A REFERENCE: 9.2(8), 3.4(4)

PROBLEM:

Each type definition introduces a distinct type, but there is no facility to guarantee that the resulting type domain is fully orthogonal in the task(s) domain. This can only be defined in the presence of fully mutative semantic operators, but the current language disallows mutation of types. This inability for orthogonal conceptual modeling in abstract spaces can be the greatest single impediment to knowledge representation in Ada.

IMPORTANCE:

(identified as a **ESSENTIAL** in AI community, **ADMINISTRATIVE** in the general community)

This has been identified in a number of studies involving issues of abstraction, representation and dynamic conceptual modeling, generically characterized as inadequate abstraction facilities. As a weakness in the language for AI use, it appears impossible to address without radical modification to the language. Yet, some creative way to increase the robustness of the abstraction mechanism **MUST** be found, or Ada will be disallowed for use by a growing segment of users.

CURRENT WORKAROUNDS:

If a good "fix" is not found, ad hoc methods (annotations) will proliferate and the standard will be effectively compromised. Currently, annotation to an expanded syntax is the only workaround.

POSSIBLE SOLUTIONS:

Full allowance of mutation will compromise other aspects of the language, but not if a simple modification is made to the rules governing type/subtype derivation. I propose that the derivation mechanism allow differing characteristics, including constraints to issue to subtypes. While this is less than optimum for an AI user, it provides a semantic primitive set to be implicitly specified which can be used (with some difficulty) for full metalevel inferencing.

The penalty is small for compiling to single-strand machines, requiring simply another multidimensional state table. Implications for true multiprocessing are more difficult, but less so than other more imminent problems.

TASKING SEMANTICS

DATE: April 21, 1989

NAME: Stewart French

ADDRESS: Texas Instruments
P.O. Box 869305 MS 8435
Plano, TX 75086

TELEPHONE: (214) 575-3522

ANSI/MIL-STD-1815A REFERENCE: 9.3 - 9.8 (ALL)

PROBLEM:

Tasking semantics incur high overhead and do not allow application programmers to control a timeline.

IMPORTANCE: IMPORTANT

CONSEQUENCES:

Vendors will continue to develop their own non-portable interfaces to their individual runtimes that allow users to circumvent the usual tasking semantics in favor of faster mechanisms for implementing tasking and tasking related constructs. The present tasking model is too complex, inflexible, and costly in terms of performance.

CURRENT WORKAROUNDS:

Non-portable calls to customized runtime execs.

POSSIBLE SOLUTIONS:

TERMINATION OF TASKS

DATE: December 5, 1988

NAME: David Brookman

ADDRESS: Magnovox Electronic Systems Company
1313 Production Road
Department 542
Fort Wayne, IN 46808

TELEPHONE: (219) 429-4440
E-mail: CONTR22ONOSC-TECR.Arpa

ANSI/MIL-STD-1815A REFERENCE: 9.4(13), 9.4(8)

PROBLEM:

Tasks declared in library unit packages are not required to terminate using the terminate alternative.

IMPORTANCE: IMPORTANT

This makes the use of the terminate alternative impossible with tasks declared in library unit packages.

CURRENT WORKAROUNDS:

The task can be forced to terminate by providing a terminate rendezvous. This rendezvous would cause the task to exit its outermost loop. Also system services could be used to terminate the process. It seems that system services are the most practical method for stopping these tasks. This is clearly not portable.

POSSIBLE SOLUTIONS:

Require tasks declared in library unit package to terminate when the following conditions are met:

1. The task is waiting on a terminate alternative.
2. The main program has completed its execution.
3. Each task within the system is either already terminated or waiting on an open terminate alternative.

ALTERNATE ADA TASK SCHEDULING

DATE: October 3, 1988

NAME: Kent Power

ADDRESS: Boeing Military Airplanes
P.O. Box 7730, MS K31-26
Wichita, KS 67277-7730

TELEPHONE: (316) 526-7235

ANSI/MIL-STD-1815A REFERENCE: 9.5(15)

PROBLEM:

The ability to employ scheduling algorithms for Ada tasking that are different from the current priority model would have benefits for embedded real-time systems. Hard deadline environments would be able to use algorithms which maximize usage of resources [1]. Multi-processor environments would be able to tailor scheduling to best use their resources. Embedded systems which require mode change or reconfiguration to recover from resource loss would be able to suspend, resume, stop, start, and restart tasks in a timely, orderly manner. Putting such flexibility in the language would enhance reuse and portability for real-time embedded software by making performance more uniform, and would accommodate future scheduling algorithms developed to meet needs not now apparent.

SPECIFIC REQUIREMENT:

The user shall be able to control the method by which tasks are scheduled.

IMPORTANCE:

In the absence of supporting this requirement, real-time projects which determine that current Ada tasking is not sufficient will develop ad hoc schedulers unique to each project and compiler vendor. For instance, requirements to recover from resource loss due to failure or battle damage must be met outside the bounds of the Ada language; each project with reconfiguration requirements must come up with a means of cleanly stopping tasks and restarting them (possibly on different processors).

It is obvious that reuse and portability will suffer since applications developed in the environment for one project with unique recovery algorithms and scheduling algorithms are not likely to perform identically in the environment of another project. Additionally, problems arising in the past from project-unique scheduling in the areas of maintenance, robustness, and intractability of anomalies will continue.

CURRENT WORKAROUNDS:

There are probably more proposed alternatives to Ada tasking than to any other language feature. It is beyond the scope of this revision request to provide a comprehensive survey of such alternatives. The intent, rather, is to point out the advantages of scheduling flexibility to the embedded systems represented by the JIAWG. By providing scheduling primitives to allow projects to construct scheduling algorithms that meet their requirements, projects using Ada are kept under the Ada "umbrella," so that the benefits of Ada accrue (such as reuse, portability, or ease of maintenance).

POSSIBLE SOLUTIONS:

The Ada Runtime Environment Working Group (ARTEWG) of SIGAda has produced a paper entitled "A Model Runtime System Interface for Ada" [2] which addresses some of the requirements presented above. However, as the ARTEWG has worked within the confines of 1815A, the paper does not address all primitives which would be necessary to achieve scheduling flexibility. Another paper from this group is "A Catalog of Interface Features and Options for the Ada Runtime Environment," [3] which addresses many of the suggested implementation's features below.

A model from Professor Elrad of Illinois Institute of Technology has been submitted to the JIAWG for consideration [4]. However, it is not clear from the model whether all the requirements above would be met (e.g., defining critical code sections).

An ideal solution would be one which allowed the following:

1. Upward compatibility from 1815A.
2. Selection of different predefined scheduling algorithms, such as stable rate-monotonic, deferrable servers [5], "complete" prioritized tasking (the 1815A model with prioritized queues and selects), or the current tasking model (perhaps the default algorithm).
3. Provision of a collection of primitives to implement user-defined algorithms.

Such a solution would allow embedded programs to meet much more of their requirements within Ada, and so enjoy more of the benefits which use of Ada brings.

An implementation might include the following capabilities accessible to the user:

1. User-definable task characteristics;
2. Access to characteristics of tasks which are queued at an accept or a select alternative (e.g., priorities, deadlines, and user-defined characteristics);
3. Selection of which call to accept at accepts;
4. Selection of which select alternative to choose (and of which call to a given select alternative);
5. Control of task priorities at run-time;
6. Halting of tasks in an orderly fashion such that cleanup operations may take place;
7. Starting and restarting of tasks;
8. Suspending and resuming tasks;
9. Passing initialization parameters to tasks; and
10. Starting and ending critical code sections which are not preemptable.

DIFFICULTIES TO BE CONSIDERED:

One difficulty is what the lowest granularity of scope for scheduling algorithms ought to be - for instance, the task, the program, or the processor. Some algorithms may co-exist (although perhaps uneasily); the 1815A tasking model can co-exist with the "complete" prioritized tasking described in [6], where task priorities are used at accepts and selects. Other algorithms which demand support from hardware (rate-monotonic, for example, in order to achieve drift-free periods) might not be compatible with other algorithms that need timing support.

Limitations of hardware ought to be considered as well. For instance, the ability to support two fundamentally different drift-free periods (such as 20 milliseconds and 16.125 milliseconds) ought not to be required (since in the absence of special hardware, this is unachievable), nor should it be prohibited (since a project may provide interrupts to achieve this).

Cornhill, et al. [1] give "an integrated approach to management of all critical system resources" as a requirement for embedded real-time scheduling, and use input-output as an example of such integrated management.

While it is true that scheduling and input-output must be integrated for a given project, in the absence of any standard controlling (for example) bus interface units, it is very difficult to see how the ARM could hope to establish requirements for the management of input-output. It is reasonable to expect the language to contain requirements which support the management of critical resources, however.

REFERENCES:

- [1] Cornhill, D., Sha, L., Lehoczky, J., Rajkumar, R., and Tokuda, H., "Limitations of Ada for Real-Time Scheduling," Proceedings of the First International Workshop on Real-Time Ada Issues, Moretonhampstead, Devon, U.K., 1987.
- [2] "A Model Runtime Environment Interface for Ada," version 1.4, Ada Runtime Environment Working Group, MRTEI Subgroup, Association for Computing Machinery, Special Interest Group for Ada, release date 2 Feb, 1988.
- [3] "A Catalog of Interface Features and Options for the Ada Runtime Environment," version 2.0, Ada Runtime Environment Working Group, Interfaces Subgroup, Association for Computing Machinery, Special Interest Group for Ada, release date Dec, 1987.
- [4] Elrad, T., "A Comprehensive Scheduling Controls [sic] for Ada Tasking," communication to Mr. Michael Mills, USAF, ASD-AFALC/AXTS, Wright-Patterson AFB, OH, 45433, September, 1988. A copy of this paper is submitted as supplementary material.
- [5] Sha, L., Lehoczky, J., and Rajkumar, R., "Task Scheduling in Distributed Real-Time Systems," Proceedings of IECON '87: 1987 International Conference on Industrial Electronics, Control, and Instrumentation.
- [6] Ada 9X Revision Request, "Use of Task Priorities in Accept and Select Statements," Kent Power, dated 10 Oct 1988.

USE OF TASK PRIORITIES IN ACCEPT AND SELECT STATEMENTS

DATE: October 3, 1988

NAME: Kent Power

ADDRESS: Boeing Military Airplanes
P.O. Box 7730, MS K31-26
Wichita, KS 67277-7730

TELEPHONE: (316) 526-7235

ANSI/MIL-STD-1815A REFERENCE: 9.5(15)

PROBLEM:

The design of real-time software often includes the use of simple priorities to determine which task, of those eligible to run, will run. Such an approach to software design uses task starvation as a tool to accomplish mission objectives, particularly when processing resources are degraded. However, it is necessary that prioritization be used whenever a choice is made which impacts which tasks are ready to execute. Examples in current Ada tasking where prioritization is not Used include taking calls off an accept queue, and choosing among open alternatives in a select statement.

SPECIFIC REQUIREMENT:

Whenever a choice is to be made within Ada tasking that affects which tasks will be eligible to execute, Ada task priorities must be used to make the choice.

IMPORTANCE:

In the absence of supporting this requirement, real-time projects will either constrain design in non-optimal ways, incur the overhead of workarounds similar to that presented above, or (most likely) sacrifice portability for vendor-unique implementations of prioritization.

CURRENT WORKAROUNDS:

Multiple tasks awaiting an accept is a situation where the choice of which call to take from the accept queue affects which tasks will be ready to execute at a later time. Since the ARM specifies that the choice of call to take from an accept queue be according to first-in, first-out (FIFO), a higher priority task which calls an entry after a lower priority task may be blocked from running.

Consider a handler for a system resource which accepts requests from a number of users. The handler is implemented as an Ada task. The users are tasks of different priorities from different applications, so that a task's priority reflects, among other things, its mission importance. Assume that the design of the software is such that the handler itself has a higher priority than that of any of its users (a traditional Handler design).

Suppose that the handler is paused awaiting the results of input-output for a user. If user Low_Priority makes a call to the handler, and then user High_Priority makes a call, the request from Low_Priority must be accepted first, according to 9.5 (15), when the handler returns to its accept body. If the handler is written in a straightforward way, user Low_Priority is eligible to execute before user High_Priority since its

request is accepted and serviced first.

One possible workaround is for the handler to continue to accept requests from its accept body until no more are found, using conditional accepts, and then internally prioritize all requests that are found.

```
package Handler is
  task Handler_Task is
    entry Receive_Requests;
  end Handler_Task;
end Handler;
package body Handler is
  task body Handler_Task is
  begin
    Service_Requests : loop
      accept Receive_Requests;
      --put request into internally prioritized queue
    loop
      select
        accept Receive_Requests;
        --put request into internally prioritized queue
      else
        exit;
      end select;
    end loop;
    --service requests
  end loop Service_Requests;
end Handler_Task;
end Handler;
```

Problems with this workaround include 1) Overhead incurred, and 2) Maintenance of a separate copy of task priorities visible to the handler, since the priority of an Ada task is not available to the handler or the task.

Another workaround is to implement non-Ada task prioritization with a particular Ada vendor, as urged by Robert Eachus. This approach meets all requirements for a given project, but there are obvious problems with portability.

Another situation where task prioritization ought to be used is in choosing an open alternative on a select statement, where the ARM specifies only an arbitrary selection (ARM section 9.7.1 (6)). While this leaves vendors free to use priorities in choosing an alternative (which some vendors do), lack of specification degrades portability and reuse.

POSSIBLE SOLUTIONS:

The obvious solution for accept queues is to require that calls to an accept are taken off the queue by priority, and FIFO within a priority level. A pragma might be used to indicate that prioritization for accepts is desired.

Another possible solution is to relax the ARM and not specify the manner in which calls are taken off an accept queue, in a manner similar to the current ARM specification for choosing select alternative. This solution has portability and reuse drawbacks.

For choosing select alternatives, reuse and portability would be enhanced if the calling task with the highest priority among all tasks queued on open select alternatives were the task to be taken.

Implementing the above solutions would also go a long way in solving the difficulties that priority inversion presents to hard deadline scheduling algorithms, and would obviate much of the need for priority inheritance as a solution to priority inversion. The only case where low priority tasks will be serviced before high priority tasks is when a high priority task makes a call to an entry immediately after a low priority request is received. This behavior is identical to that of many fielded avionics systems; problems attributable to priority inversion have not been observed.

DIFFICULTIES TO BE CONSIDERED:

One consideration is whether upward compatibility between 1215A and 9X is to be maintained.

REFERENCES:

- [1] Cornhill, D., Sha, L., Lehoczky, J., Rajkumar, R., and Tokuda, H., "Limitations of Ada for Real-Time Scheduling," Proceedings of the First International Workshop on Real-time Ada Issues, Moretonhampstead, Devon, U.K., 1987.

CONTROL OF CLOCK SPEED AND TASK DISPATCH RATE**DATE:** January 26, 1989**NAME:** Gilbert T. Williams**ADDRESS:** Westinghouse Electric Corporation
Oceanic Division / Cleveland Operation
18901 Euclid Avenue
Cleveland, OH 44117**TELEPHONE:** (216) 486-8300 ext. 1069**ANSI/MIL-STD-1815A REFERENCE:** 9.6, 9.7, 13.**PROBLEM:**

There is no language-defined way to alter the clock speed (task dispatch rate) for controlling the execution of multi-tasking programs in a faster (or slower) than real-time simulation environment. Currently, all tasks that use delay and rendezvous constructs depend on the actual real-time system clock to control execution. A language-defined method for altering this behavior would improve the portability of code to an embedded target system and eliminate the need for a special purpose simulation executive.

IMPORTANCE: ESSENTIAL

We believe that the ability to simulate a program on a host machine prior to hardware selection enables the system engineer to more effectively size the hardware and thus manage risk. The addition of the proposed capability to the existing Ada standard should provide more general acceptance of the "Software First" approach to system development. In addition, being able to operate at speeds different than real-time would provide better stress testing of multi-tasking programs.

If this suggestion is not implemented, other Ada users interested in simulation will need to take an approach similar to ours (outlined below) to provide this very desirable capability.

CURRENT WORKAROUNDS:

For our in-house simulation system, we designed a custom pseudo-real-time modeling executive to provide task control during simulation. Although we could have patched the existing DEC Ada run-time system, this would have made it virtually impossible to port the software to other target machines. As it is, it is possible to port the software by rewriting the lowest-level context switching routines of the modeling executive as appropriate for the new target.

POSSIBLE SOLUTIONS:

- Provide a pragma to allow a user-written executive to be called from the Ada tasking structures, including delay and select statements (both entry calls and accept constructs). This pragma would specify the entry points in the user executive to be called when tasking activity is requested by the application program.
- Provide a PRAGMA to allow user-written system clock interface to be utilized by the implementation's run-time system.

- Require that each implementation document the internals of the Ada tasking structure in an Appendix.
- Provide a section in chapter 13 that defines how an implementation controls the clock.

CONFIGURING CALENDAR.CLOCK IMPLEMENTATION

DATE: June 7, 1989

NAME: Richard D. Powers (ACM Special Interest Group on Ada, Ada Runtime Environment Working Group)

ADDRESS: Texas Instruments
P.O. Box 869305
M/S 8435
Plano, TX 75086

TELEPHONE: (214) 575-3562
E-mail: (ARPAnet) rpowers@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 9.6, 13.7

PROBLEM:

The language should provide a standard way for application builders to provide the configuration-dependent information upon which the implementation of CALENDAR.CLOCK depends.

SPECIFIC REQUIREMENT/SOLUTION CRITERIA:

For many embedded applications the exact configuration of timer devices varies with the target hardware. It is therefore impossible for a compiler implementor to know the details of the hardware timer(s). A mechanism is needed for the application builder to provide these details to the RTE.

An important special case is where the rate of the hardware clock is user-adjustable. In this case the application builder needs to be able to specify the actual time interval between hardware clock ticks. This value does not need to be changed while the system is running, but solutions that require the clock update interval be specified at compile-time are inferior to those that permit this value to be specified later.

IMPORTANCE:

Applications builders will have to modify the source to the clock interrupt handler on the target system to reflect the actual clock tick interval. Otherwise, the implementation provided value may be arbitrary, and CALENDAR.CLOCK will advance at an incorrect rate.

CURRENT WORKAROUNDS:

Currently, application builders are required to obtain and modify some portion of the Ada RunTime Environment (RTE) code. The interface between the user-modified portion and remainder of the RTE is nonstandard. Modifying this interface is wasteful, requires detailed understanding of the RTE, and requires hard-to-obtain information that is specific to each Ada implementation. This results in duplication of effort, and a possibility of errors in each reimplementing of a low-level piece of RTE.

POSSIBLE SOLUTIONS:

The language could provide a pragma which sets the value of SYSTEM.TICK (analogous to pragma STORAGE_UNIT), with the same restrictions as pragma STORAGE_UNIT and the other pragmas in

Section 13.7. Alternatively, the language could require support for recompilation of package SYSTEM with a new value for SYSTEM.TICK.

A more general solution is to define a standard interface between the RTE and the user-modified clock support routine.

DIFFICULTIES TO BE CONSIDERED:

Some compilers may have values related to DURATION and CALENDAR.CLOCK "hard-coded" into the compiler. Such compilers might need modification to accept a system configuration value. The differences between hardware clocks might be difficult to specify across RTE implementations. These hardware differences could make specifying a standard interface difficult.

DELAY UNTIL**DATE:** June 7, 1989**NAME:** Richard D. Powers (ACM Special Interest Group on Ada, Ada RunTime Environment Working Group)**ADDRESS:** Texas Instruments
P.O. Box 869305
M/S 8435
Plano, TX 75086**TELEPHONE:** (214) 575-3562
E-mail: (ARPAnet) rpowers@ajpo.sei.cmu.edu**ANSI/MIL-STD-1815A REFERENCE:** 9.6**PROBLEM:**

Applications require the ability to schedule delay expiration at an absolute (i.e. Calendar) time.

SPECIFIC REQUIREMENT/SOLUTION CRITERIA:

Some applications need the ability to schedule a task to be made eligible for execution at a specific calendar time. Currently, Ada supports delaying for a relative time, but no reliable mechanism is given for delaying until a Calendar time.

If a task requests to delay until some time, T, it should be guaranteed that within some implementation-defined tolerance, the task will become eligible to execute as soon as `CALENDAR.CLOCK` is greater than or equal to T. This property should hold, even if the clock is adjusted forward or backward.

IMPORTANCE:

The lack of support for this feature leads to problems that are very difficult to uncover. This is especially true when code like the above example is used. Such code will often work a large percentage of the time, and only fail on that rare occasion where an interrupt arrives during execution of the described critical region, or when the clock is adjusted.

CURRENT WORKAROUNDS:

A system may need to have a task that runs at midnight every night to record information about the processing that has occurred, etc. In current Ada, the task would be written as follows:

```

task body AT_MIDNIGHT is
  ONE_DAY : constant CALENDAR.DAY_DURATION := 86_400.0;
  NEXT_TIME : CALENDAR.TIME;
begin
  NEXT_TIME := CALENDAR.CLOCK;
  NEXT_TIME := CALENDAR.TIME_OF(YEAR => CALENDAR.YEAR(NEXT_TIME),
                                MONTH => CALENDAR.MONTH(NEXT_TIME),
                                DAY   => CALENDAR.DAY(NEXT_TIME),

```

```

                                SECONDS => 0.0);
NEXT_TIME := CALENDAR."+"(NEXT_TIME,ONE_DAY);
loop
  delay CALENDAR."-"(NEXT_TIME,
                    CALENDAR.CLOCK -- (*)
                    );
  -- sample input
  NEXT_TIME := CALENDAR."+"(NEXT_TIME,ONE_DAY);
end loop;
end AT_MIDNIGHT;

```

There are two problems with this code:

- 1) The delay statement, including the "-" operation, is not guaranteed to be atomic. Therefore, if an interrupt occurs during the execution of this statement, the wake-up time can be later than was intended. There exists a critical region between the moment the task obtains a value from CALENDAR.CLOCK at point (*) and the moment the Ada runtime system schedules the wake-up. If the task is preempted during this time, the runtime system will schedule the delay relative to the time at which the task resumes execution. This may be enough later than was intended that the application fails.
- 2) If the system clock is adjusted by an external input (e.g. for daylight savings time, to correct for clock drift among multiple processors, or to synchronize with other sites), the delay value may not be adjusted. In fact, for a typical relative delay, the delay time should not be adjusted.

POSSIBLE SOLUTIONS:

One possible solution is to support a new form of the delay statement that takes an expression of type CALENDAR.TIME. Given such a delay statement, the above example could be written:

```

task body AT_MIDNIGHT is
  ONE_DAY : constant CALENDAR.DAY_DURATION := 86_400.0;
  NEXT_TIME : CALENDAR.TIME;
begin
  NEXT_TIME := CALENDAR.CLOCK;
  NEXT_TIME := CALF AR.TIME_OF(YEAR => CALENDAR.YEAR(NEXT_TIME),
                              MONTH => CALENDAR.MONTH(NEXT_TIME),
                              DAY => CALENDAR.DAY(NEXT_TIME),
                              SECONDS => 0.0);
  NEXT_TIME := CALENDAR."+"(NEXT_TIME,ONE_DAY);
loop
  delay NEXT_TIME; -- argument of type CALENDAR.TIME
  -- sample input
  NEXT_TIME := CALENDAR."+"(NEXT_TIME,ONE_DAY);
end loop;
end AT_MIDNIGHT;

```

DIFFICULTIES TO BE CONSIDERED:

Forcing CALENDAR's math functions to be non-interruptible is not a sufficient solution; as there still exists the possibility of preemption between the evaluation of the simple expression for the delay statement and the RTE scheduling of the delay.

The addition of an absolute delay capability adds complexity to the handling of all language features that support delays (e.g. select statements). It may also complicate the handling of clock synchronization on embedded systems that support such synchronization. (For instance, a system may get a time "update" from an external source, and then have to adjust all tasks that are "delaying until" appropriately.)

Allowing an expression of type TIME in a delay statement would probably require moving the declaration of type TIME into package STANDARD, for the same reasons type DURATION is declared there now.

ASYNCHRONOUS TRANSFER OF CONTROL

DATE: April 20, 1989

NAME: S. Tucker Taft

ADDRESS: Intermetrics, Inc.
733 Concord Avenue
Cambridge, MA 02138

TELEPHONE: (617) 661-1840
E-mail: stt@inmet.inmet.com
E-mail: uunet!inmet!stt

ANSI/MIL-STD-1815A REFERENCE: 9.7.1

PROBLEM:

To satisfy perceived requirements for asynchronous transfer of control [Ada letters special edition, Vol VII.7 Fall 88 -- 2nd Int'l Wkshp on RT Ada Issues -- hereafter IWRT2], we propose the provision for an asynchronous entry call/selective wait construct.

IMPORTANCE:

Non-portable outside-of-Ada mechanisms will be relied upon to provide asynchronous transfer of control when needed, or the overhead and uncertainties associated with the "abort" statement will be endured.

CURRENT WORKAROUNDS:

There are many situations where it is useful to terminate one sort of processing and continue processing in a new place, based on an external/terminal interrupt, mode switch, etc. Currently interrupts may be bound to task entries, but there is no action which can be taken within the corresponding accept body to terminate ongoing processing other than the abort statement. It was felt by most of the participants of [IWRT2] that this approach was unacceptably expensive and inadequately controllable. The approach most widely recommended in [IWRT2] was asynchronous exceptions. However, many problems were identified with this approach, requiring an number of new concepts such as enabling/disabling exceptions, new classes of exceptions, queuing rules, etc. We feel that an asynchronous entry call/selective wait construct solves many of these problems while introducing fewer new concepts into the language.

POSSIBLE SOLUTIONS:

Provide a new selective wait construct:

```
select
  select_alternative
{ or
  select_alternative}
and
  sequence_of_statements
end select;
```

The semantic of this construct are that normal processing is performed on the select alternative guards to determine which should be "open." Selection of one such open alternative takes place immediately if a rendezvous is possible, or if a delay alternative of less than or equal to zero seconds is open. Otherwise, the sequence_of_statements begin execution. If the sequence_of_statements completes execution, then the select alternatives are closed.

If prior to completion of the sequence_of_statements and outside of any nested rendezvous (either accept or entry call), a delay alternative has expired, or an open accept alternative has a caller, then the sequence_of_statements is abandoned, and an asynchronous transfer of control takes place to the appropriate open select alternative. This abandonment takes place no later than the next synchronization point, but it is the intent that any ongoing computation (outside of a rendezvous) be preempted.

If the same entry is made open via a nested selective wait or accept statement, then the inner construct takes precedence.

Nested rendezvous are protected from preemption to prevent corruption of protected data structures and undesirable side effects in the calling task. Note however, that within a nested accept it is possible to further nest an asynchronous selective wait thereby again allowing for nested asynchronous transfer of control.

Here are some examples of use:

```

loop -- Main command loop for a command interpreter
  select
    accept Terminal_Interrupt;
    Put_Line("Interrupted");
  and
    -- This will be abandoned upon terminal interrupt
    Put_Line("-> ");
    Get_Line(Command, Last);
    Process_Command(Command(1..Last));
  end select;
end loop;

```

```

-----
select -- Perform time-limited calculation
  delay 5.0;
  Put_Line("Calculation does not converge");
and
  -- This calculation should finish in 5.0 seconds
  -- if not, it is assumed to diverge
  Horribly_Complicated_Recursive_Function(X, Y);
end select;

```

PRIORITY SELECT**DATE:** April 21, 1989**NAME:** Richard Powers**ADDRESS:** Texas Instruments
P.O. Box 869305 MS 8435
Plano, TX 75086**TELEPHONE:** (214) 575-3562**ANSI/MIL-STD-1815A REFERENCE:** 9.7.1(6)**PROBLEM:**

No select based on task priority. A select with multiple accept statements does not schedule the entry tasks based on priority. In addition, the entry queues are not priority queues.

IMPORTANCE: IMPORTANT**CONSEQUENCES:** High priority tasks do not receive priority attention.**CURRENT WORKAROUNDS:****POSSIBLE SOLUTIONS:**

TERMINATE NOT USED

DATE: April 21, 1989

NAME: Stewart French

ADDRESS: Texas Instruments
P.O. Box 869305 MS 8435
Plano, TX 75086

TELEPHONE: (214) 575-3522

ANSI/MIL-STD-1815A REFERENCE: 9.7.1(10)

PROBLEM:

The terminate alternative adds complexity to the Ada runtime, but affords very little in program control; construct not frequently used.

IMPORTANCE: IMPORTANT

CONSEQUENCES:

CURRENT WORKAROUNDS:

POSSIBLE SOLUTIONS:

DYNAMIC PRIORITIES FOR TASKS

DATE: October 4, 1988

NAME: Chuck Roark

ADDRESS: Texas Instruments
P.O. Box 869305, MS 8435
Plano, TX 75086

TELEPHONE: (214) 575-3537
E-mail: ROARK@EG.TI.COM

ANSI/MIL-STD-1815A REFERENCE: 9.8

PROBLEM:

Real-time, integrated systems require fast initialization, especially during reconfiguration. Current Ada has activation of a task executing at the priority of the task being activated during which the parent of the task is suspended. Hence, under current Ada rules, a task which is being activated and which is low priority can have a major impact on initialization times.

SPECIFIC REQUIREMENT:

Need mechanism to allow dynamic software control over time it takes to initialize/activate tasks, with capability to minimize the time a parent task is suspended. Need the abilities to (1) allow task activation to occur at any urgency and (2) allow a task's urgency to be different than its activation urgency after it has been activated. In particular, to support fast initialization, it must be possible for a task's activation urgency to be higher than any task eligible for execution (other than itself or other tasks being activated) and for the task's urgency to be lowered following its activation (even if this lowering occurs due to code executed by the activated task).

IMPORTANCE:

A work around already exists as stated in the previous paragraph. However, adding this capability to the language would allow this approach to be portable between different Ada implementations.

CURRENT WORKAROUNDS:

Programs such as LHX have stringent run-time reconfiguration requirements - .25 seconds for critical functions in LHX. The summary states the effect that current Ada rules can have on fast initialization. By allowing dynamic priorities, current Ada activation rules will support fast initialization. Pragma priority can be used to specify the priority to be used when a task is activated. This will support fast task execution where this is desired since a high priority can be used during activation. Following activation the beginning of task code can make a subprogram call to a "change_priority" routine to lower its priority to its intended execution priority.

Current workarounds are to supply a change_priority routine (e.g., Tartan Labs 1750A Ada compiler) as an extension to the Ada runtime.

POSSIBLE SOLUTIONS:

Support dynamic priorities as discussed above.

DIFFICULTIES TO BE CONSIDERED:

None foreseen.

REFERENCES/SUPPORTING MATERIAL:

Reference Tartan ARTClient package and ARTEWG CIFO documentation for approaches to supporting dynamic priorities.

MODIFICATION OF TASK PRIORITIES DURING EXECUTION**DATE:** December 5, 1988**NAME:** David Brookman**ADDRESS:** Magnovox Electronic Systems Company
1313 Production Road
Department 542
Fort Wayne, IN 46808**TELEPHONE:** (219) 429-4440
E-mail: CONTR22ONOSC-TECR.Arpa**ANSI/MIL-STD-1815A REFERENCE:** 2.8**PROBLEM:**

It is not possible to set task priorities during execution. Sometimes the relative importance of certain functions may change during the execution of a program. It may be desirable to change the priority of the task at that time.

IMPORTANCE: IMPORTANT

This would make task priorities more useful. Failure to fix this problem would make task priorities ineffective for applications where the relative importance of tasks changes during execution.

CURRENT WORKAROUNDS: NONE**POSSIBLE SOLUTIONS:**

TASK SCHEDULING**DATE:** December 5, 1988**NAME:** David Brookman**ADDRESS:** Magnovox Electronic Systems Company
1313 Production Road
Department 542
Fort Wayne, IN 46808**TELEPHONE:** (219) 429-4440
E-mail: CONTR22ONOSC-TECR.Arpa**ANSI/MIL-STD-1815A REFERENCE:** 9.8**PROBLEM:**

A high priority task may be suspended because it needs to rendezvous with a low priority task. That low priority task does not get scheduled promptly because of its priority. However this causes the high priority task to be suspended also.

IMPORTANCE: IMPORTANT

This problem makes the use of task priorities extremely difficult to apply correctly in a large system. It limits the ability to use task priorities to improve throughput in a system.

CURRENT WORKAROUNDS:

Design the code so that high priority tasks are never waiting for low priority ones. This is difficult or impossible to achieve for complicated systems.

POSSIBLE SOLUTIONS:

When a high priority task is suspended because it needs to rendezvous with a lower priority task, that lower priority task should inherit the priority of the high priority task. When the rendezvous is complete, the task should be returned to its original priority.

TASK PRIORITIES
(ADA-UK/012)**DATE:** February 9, 1989**NAME:** JGP Barnes (from material supplied by members of Ada-UK)**ADDRESS:** Alsys Ltd
Newtown Road
Henley-on-Thames
Oxon
RG9 1EN
UK**TELEPHONE:** +44-491-579090**ISO 8652 ANSI/MIL-STD 1815A REFERENCE:** 9.8**PROBLEM:**

The Ada priority system has proved quite inadequate for the needs of certain classes of hard real-time embedded systems. These are applications where a high degree of responsiveness is required.

For example, there is a major conflict between the fifo mechanism prescribed for the entry queue and the need for the highest priority task to proceed wherever possible.

A more precise definition of a priority system is required. It should also be noted that any system really satisfying the needs of embedded applications is unlikely to be implementable on top of development operating systems such as UNIX.

This problem and potential solutions are well documented elsewhere but in summary, the key problems are: task entries should be queued on a priority basis (and presumably fifo within priorities) accept alternatives in a select statement should be on a priority basis and not just arbitrary (and presumably arbitrary within priority) a rendezvous should inherit the priority of a higher priority task joining the entry queue.

In addition, dynamic alteration of priorities should be allowed so that the user can implement special systems where required.

Serious consideration should then be given to removing entry families which are an ad-hoc and non-orthogonal historic encrustation.

IMPORTANCE: ESSENTIAL

This is probably the most important change required to Ada. Ada is currently not being used by the hard real-time community or else various workarounds are being used in order to avoid the Ada scheduling. Ada was specifically designed for embedded systems and it is ironic that this aspect is being avoided.

CURRENT WORKAROUNDS:

Current users are either using their own executive sitting on top of the Ada run-time system or bypassing the Ada tasking model altogether. This is either inefficient or non-portable. In either event it is wasteful of resources.

POSSIBLE SOLUTIONS:

As outlined above. It might be a good idea to introduce a distinct package containing the subtype `PRIORITY` and subprograms such as `SET_PRIORITY` and `MY_PRIORITY` rather than having them inside `SYSTEM`. Such a package might be a secondary standard.

This topic has been studied in some depth over the years and in particular at the two International Real-time Workshops sponsored by AdaUK and SigAda. The proceedings of these workshops contain a number of relevant papers.

REVOKE AI-00594/02**DATE:** June 7, 1989**NAME:** Roger Racine (ACM Special Interest Group on Ada, Ada Runtime Environment Working Group)**ADDRESS:** MS 4B
C.S. Draper Laboratory
555 Technology Sq.
Cambridge, MA 02139**TELEPHONE:** (617) 258-2489
E-mail: rracine@ajpo.sei.cmu.edu**ANSI/MIL-STD-1815A REFERENCE:** 9.8**PROBLEM:**

AI-00594/02 allows implementors far too much freedom in the interpretation of task priorities. This AI must be revoked and the word "sensibly" in section 9.8 of the Ada Reference Manual (ARM) must be deleted. While it is understood that the priority mechanism of current Ada causes problems with certain applications, the solution given in AI-00594/02 must not persist as the allowed solution.

SPECIFIC REQUIREMENT/SOLUTION CRITERIA:

If two tasks are both eligible for execution on the same processor, then it cannot be the case that the task with the lower priority is executing while the task with the higher priority is not. AI-00594/02 must not persist as the only solution to this complex problem.

IMPORTANCE:

Implementations, after AI-00032, were forced to be similar in their semantics. AI-00594/02 gives implementations freedom to implement schedulers in a totally non-portable manner, as long as they can describe an application which justifies a particular implementation. Real-time code has become non-portable due to this AI.

CURRENT WORKAROUNDS:

AI-00594/02 allows implementations to do the following:

Task A1 is a high priority task which is blocked.
Task A2 is a medium priority task which is blocked.
Task A3 is ready to execute at low priority.

Task B1 is ready to execute at high priority.
Task B2 is ready to execute at medium priority.
Task B3 is ready to execute at low priority.

A3 is allowed to run, even though B1 is ready to execute.

The AI states that this is needed for simulation of multiprocessor systems. However, since the compilation system (and the ACVC tests) can not know the nature of the application, it may allow this priority inversion in other applications, such as a hard real-time system. This is unacceptable.

POSSIBLE SOLUTIONS:

This revision request does not state that the static priority scheme specified in the ARM is sufficient. Alternatives such as priority inheritance, Rate Monotonic Scheduling, and fair scheduling for time-shared systems should be considered in the Ada 9X revision process. A mechanism to permit applications to select among scheduling methods should be explicitly provided in the language definition, so that programs which rely on a specific scheduling mechanism are portable.

The language should explicitly state that within a processor's scheduler, the highest priority task which is eligible for execution must run. The priority referred to is the priority a task has at the time the dispatcher is invoked, thus allowing some form of dynamic priorities. However, if static priorities exist, the application must be able to rely on the priorities given to its tasks.

REFERENCES/SUPPORTING MATERIAL:

AI-00594/02
AI-00032

PRIORITY ENTRY QUEUING**DATE:** April 21, 1989**NAME:** Richard Powers**ADDRESS:** Texas Instruments
P.O. Box 869305 MS 8435
Plano, TX 75086**TELEPHONE:** (214) 575-3522**ANSI/MIL-STD-1815A REFERENCE:** 9.8(6)**PROBLEM:**

Current LRM does not allow entry queue position to be based on priority. Tasks should be queued by priority or FIFO, depending on application.

IMPORTANCE: IMPORTANT**CONSEQUENCES:** Priority inversion.**CURRENT WORKAROUNDS:**

Having RTS "pretend" task is not on entry queue when it actually is.

POSSIBLE SOLUTIONS:

ALLOW MODIFIABLE PRIORITIES FOR TASKS**DATE:** June 7, 1989**NAME:** Michael Victor (ACM Special Interest Group on Ada, Ada Runtime Environment Working Group)**ADDRESS:** Missile Systems Division
Raytheon Company
Hartwell Avenue SWA 6-2
Bedford, MA 01730**TELEPHONE:** (617) 270-1580
E-mail: victorm@ajpo.sei.cmu.edu**ANSI/MIL-STD-1815A REFERENCE:** 9.8(6)**PROBLEM:**

The request for user modifiable priorities is driven by at least two categories of requirements - mode change and degradation in computing resources.

SPECIFIC REQUIREMENT:

The user must be capable of specifying initial priorities for any or all tasks. The language should not preclude the subsequent explicit modification of priorities by the application.

IMPORTANCE:

Applications will go outside of the language to resolve this issue.

CURRENT WORKAROUNDS:**Example 1**

Consider the following example of a mode change requirement. An air defense application operates in two modes: nonthreatening and threatening. The former case applies to a low state of alert condition. In this mode the primary function of the system is to monitor its readiness for combat. To maintain a high level of confidence in the processors, a heavy load of tracks is forced by injecting artificial aircraft returns. The status monitor functions of the system (which are allowed to degrade in the event of an actual raid) must be raised in priority to assure that they are executed in favor of (or at least in conjunction with) the spurious track threads. Once the state of alert is raised, thereby causing a transition to the threatening mode, track priorities must exceed status monitoring priorities.

Example 2

Some system designs must react appropriately to a reduction in CPU availability. An application whose priority assignments are driven strictly by measures of functional importance will degrade gracefully in the event of the loss of some computing resources. In such a circumstance those low priority functions will automatically cease execution if there is insufficient CPU capacity. For example an air defense system which performs, in decreasing order of criticality, guidance, surveillance and display functions will assign priorities

accordingly. When resource degradation occurs, the display processing will naturally be sacrificed.

It is possible that priority assignment can be driven by other considerations however. Assume that under normal conditions, adequate resources are available to assure execution of all functions. Consider the case of a telemetry function which is not mission critical but must be executed in a timely fashion to provide useful data. This telemetry collection task can be assigned a high priority to be sure that it can collect and format required data. If and when the system degrades the priority of the telemetry process must be reduced since it is not mission critical.

DIFFICULTIES TO BE CONSIDERED:

It may be appropriate, for security or integrity reasons, to restrict the ability to modify priorities so that only a subset of priority levels is accessible to a task. Security reasons in this context refer to, among others, access privileges or verifiability in trusted systems.

Modifiable priorities may compound the difficulties of sustaining deterministic behavior.

Modifiable priorities may also compound the difficulties of insuring portability.

REFERENCES:

- [1] Steelman, Department of Defense Requirements for High Order Computer Programming Languages, June 1978
- [2] A Catalogue of Interface Features and Options for the Ada Runtime Environment, Version 2.0, ACM/SIGAda/ARTEWG, DEC 1987

PRAGMAS FOR TASK INTERRUPTS AND TIMED I/O**DATE:** July 21, 1989**NAME:** CAPT Dennis Lewis**ADDRESS:** RM 239 BLDG 13
AFATL/FXG
EGLIN AFB, FL 32542-5434**TELEPHONE:** (904) 882-3346**ANSI/MIL-STD-1815A REFERENCE:** 9.8.1**PROBLEM:**

The tasking model is awkward and arbitrary. Lt Marc Pitarys in the WPAFB Avionics Laboratory stated (in the 13 Feb 1989 issue of Advanced Military Computing) that to make compilers work effectively for the 1750A chip the written code must "avoid Ada tasking". At a June 1989 MacDac IRAD briefing to the AFATL/FXG Inertial Section, MacDac stated they had avoided Ada tasking in developing avionics software for a flight test program. Clearly to be of use in Avionics, the Ada tasking model needs to have less overhead. What would be especially nice would be a pragma specifying a non-maskable interrupt type task, which gives that task immediate access to the CPU. Almost all microprocessors have a NMI (non-maskable interrupt) command, and a pragma task should be written to take advantage of this command. Indeed, without this priority interrupt, tasking will never be used for critical real-time embedded applications.

IMPORTANCE: ESSENTIAL**CURRENT WORKAROUNDS:**

The current workarounds Ada software developers are using are proprietary run-time systems and libraries. The MacDac group that briefed the AFATL/FXG inertial section used the DEC run-time system VAX/ELN which has run-time routines for priority interrupts, and input-output.

POSSIBLE SOLUTIONS:

Some more definitive pragmas for tasks should be added. These pragmas would allow for urgent interrupts, and timed input-output through the Ada task model. Also the Distributed Real-Time Ada Kernal (DARK) program could become part of the Ada 9.8.1 standard.

COUNT ATTRIBUTE**DATE:** June 18, 1989**NAME:** Dr. Gertrude Levine**ADDRESS:** Fairleigh Dickinson University
1000 River Road
Teaneck, NJ 07666**TELEPHONE:** (201) 692-2020**ANSI/MIL-STD-1815A REFERENCE:** 9.9**PROBLEM:**

The use of the count attribute as a boolean guard on an entry is not secure, since the guard is not reevaluated if an entry call is abandoned.

IMPORTANCE: ESSENTIAL**CURRENT WORKAROUNDS:**

An additional rendezvous may be used instead of a count attribute in a boolean guard.

POSSIBLE SOLUTIONS:

The cost of reevaluating boolean guards if entry calls are abandoned due to time-outs or aborts is minimal. Securing the use of a count attribute for a boolean guard will simplify and expedite much code. We note that the introduction of an extra rendezvous to work around the unreliability of the count attribute may enable deadlocks states.

ABORT STATEMENT**DATE:** February 8, 1989**NAME:** Joanne Goldenberg

Gertrude Levine

ADDRESS: Giordano Associates, Inc.
44 Route 46
Pine Brook, NJ 07058Fairleigh Dickinson University
100 River Road
Teaneck, NJ 07666**TELEPHONE:**

(201) 808-8500

(201) 692-2000

ANSI/MIL-STD-1815A REFERENCE: 9.10**PROBLEM:**

The inclusion of an abort statement makes it difficult to prove anything about program states, deadlock states, etc. Because the allowable time lapse before termination is not defined by the language, the abort statement may be inappropriate for critical time functions. All space is not reclaimed after a task has aborted, so memory constraints should be considered before its extensive use. And finally, the abort statement does not take into account any hardware affected by the aborted task. It is possible that damage may occur to hardware that has been interrupted before a safe shutdown procedure is run.

Ada is designed for reliability and safe practice. Dangerous practices are always off by default, such as unchecked deallocation or unchecked conversion. Safe practices are always on by default, such as index an ' range checking. Why is abort designed differently?

IMPORTANCE: ADMINISTRATIVE

If this request is not satisfied with this revision there will be no change in the way abort is used (or not used) by the Ada community.

CURRENT WORKAROUNDS:

The most common workaround in use today is just not to use the abort command. Other possible workarounds include a rendezvous where safe shutdown is performed before the abort statement is executed.

POSSIBLE SOLUTIONS:

To effectively control the abort statement, we recommend making a few minor modifications to the language. The default abort would be turned off. This would restrict a task from being aborted when it is in a critical situation. If an abort of a protected task is attempted, a `TASKING_ERROR` exception would be generated.

Controlled usage of abort can then be implemented with `"PRAGMA ABORT_ON [(OPTIONAL TASK TYPE)]`. When `PRAGMA ABORT_ON` is used within a particular module, the task(s) within that module would be able to be aborted by any other task for which it is visible.

Further control can be exercised by restricting legal abortions to particular task types within a module. A `PRAGMA ABORT_ON (FIRST_TASK_TYPE)` placed within a procedure would enable all tasks of

FIRST_TASK_TYPE to be aborted, while other task types within that procedure are kept safe from an unexpected ending.

By adding the PRAGMA ABORT_ON facility, the programmer is given the opportunity to protect a large majority of the code which he knows to be critical, while still having the flexibility to allow aborted tasks within a controlled environment. Programmers can use time-outs or exceptions to branch into and out of abort-free modules.

The key point is that this command would allow the programmer to decide at the design phase what tasks perform critical functions and take steps to protect these tasks. Isolating a dangerous practice and restricting it to a particular module and, possibly even a particular task type within that module, allows modular programming using tasks and enforces the safe endings of particular tasks.

SHARED COMPOSITE OBJECTS

DATE: June 7, 1989

NAME: Ted Baker (ACM Special Interest Group on Ada, Ada Runtime Environment Working Group)

ADDRESS: Department of Computer Science
Florida State University
Tallahassee, FL 32306-4019

TELEPHONE: (904) 644-5452
E-mail: tbaker@ajpo.sei.cmu.edu
E-mail: baker@nu.cs.fsu.edu

ANSI/MIL-STD-1815A REFERENCE: 9.11

PROBLEM:

Some applications require shared variables that are of a composite type, but where the only references that need to be synchronization points are to scalar or access components. The pragma SHARED should be extended to apply to such objects for operations applied component-wise to the scalar or access components.

SPECIFIC REQUIREMENT/SOLUTION CRITERIA:

It must be possible to specify that specific array and record variables are shared, so that whenever a read or update operation is performed on a simple scalar or access component of such a structure it can be guaranteed to be performed atomically, and the value will be actually read from or updated in memory at that point.

IMPORTANCE:

Programs that use shared memory buffers for asynchronous operations will be erroneous, relying on compiler-specific features or luck for correct function.

CURRENT WORKAROUNDS:

The main requirement is to support data structures, such as buffer areas, that are shared between Ada programs and memory-mapped or direct-memory access hardware devices. The problem also arises with data areas shared between Ada programs and interfaced code written in other languages, especially when it may be executed asynchronously as in assembly-language interrupt handlers.

The problem that must be addressed is that optimizing compilers may suppress loads and stores of values that they perceive as "dead", or which are perceived as implementable via local cache storage or registers. The cases mentioned above are ones where it is likely to appear to an Ada compiler that a read or update operation need not be performed to memory immediately, or at all.

A good example is an array which is used as an output buffer. Suppose the array is written by the Ada program and read by a DMA output device or assembly-language device driver. Since the Ada code never

reads the values written to this buffer, a compiler might well suppress all write operations to it.

The same danger of compiler optimization arises then using cyclic (single writer, single reader, FIFO) buffers to communicate between Ada tasks (including Ada interrupt handlers). While primitive, this is a well-known, efficient, technique for safe communication between concurrent processes.

Note that the intent here is not to require any special synchronization or atomic operations. This feature need only be implemented for types of data for which the hardware supports atomic read and write operations (i.e. single words). The only intent is to force the compiler to avoid optimizations that will be unsafe for concurrent execution.

POSSIBLE SOLUTIONS:

The restriction in 9.11(10) should be reworded to state that the pragma SHARED is also allowed for a variable declared by an object declaration and whose type is an array or record type, but in this case only a read or update of an individual scalar or access component of the variable is a synchronization point, and only for the specific component to which it applies. (That is, reads and of the whole array or record are not synchronization points.)

For additional references to Section 9. of ANSI/MIL-STD-1815A, see the following sections, revision request numbers, and revision request titles in this document.

REVISION REQUEST

<u>NUMBER</u>	<u>TITLE</u>	<u>SECTION</u>
0117	PRE-ELABORATION	3
0111	FAULT TOLERANCE	5
0104	ACCESSING A TASK OUTSIDE ITS MATTER	5
0128	SUBPROGRAMS AS PARAMETERS	6
0140	PROBLEMS WITH OBJECT ORIENTED SIMULATIONS	7
0090	CONTROL OVER VISIBILITY OF TASK ENTRIES	7

**ADA 9X REVISION REQUESTS
THAT REFERENCE
ANSI/MIL-STD-1815A**

**SECTION 10. PROGRAM STRUCTURE
AND
COMPILATION ISSUES**

SECTION 10 SHOULD NOT DESCRIBE THE PROCESS OF COMPILATION, BUT RATHER THE MEANING OF PROGRAMS

DATE: March 20, 1989

NAME: T G L Lyons (from material supplied by members of Ada UK and Ada Europe
Environment Working Group)

ADDRESS: Software Sciences Limited
Meudon Avenue
Farnborough
Hnats
GU14 7NB
UK

TELEPHONE: +44 252 544321

ANSI/MIL-STD-1815A REFERENCE: 10.

PROBLEM:

Chapter 10 of the LRM is written in such a way that compiler implementations are expected to do things in a certain way (i.e., the description is procedural rather than "declarative"). This is in contrast to the rest of the LRM, which describes the effect of elements of the language, rather than the means by which those effects are achieved. In consequence, the LRM is not well adapted to different kinds of implementation (e.g., interpreters or even incremental compilers rather than compilers) and the tight coupling between an Ada compiler and its library manager generally makes it impossible to integrate an Ada compiler properly with a configuration management system or within an APSE.

Chapter 10 has many examples of specific descriptions of implementation oriented concepts. For example, it talks of "compilation units submitted to a compiler", of "the text of a program ... submitted to a compiler", of a "program library" (section 10.1 para 3) and of a "library file" (section 10.4 para 1).

The chapter has many details of a specific process. For example it defines the "order in which units can be compiled", it defines the effect of compilation failure and it defines the effect of recompilation in terms of making units obsolete and requiring these to be recompiled.

All these notions cause problems, and this is directly related to the fact that the chosen method of description is inappropriate, and should be replaced by a "declarative" approach.

Specifically, the LRM appears to require the existence of a program library and a library files. The implication of a single program library is acknowledged in the note in section 10.4, which makes it necessary to explain that there may be more than one program library. If the inappropriate notion of program library had not been introduced in the first place, then it would not have been necessary to provide this excuse. Moreover, the wording does not appear to allow multiple program libraries to be used for a compilation, such a restriction seems unnecessary, and strictly speaking presumably makes invalid any compiler which does so.

The rules for order of compilation are otiose. Consider:

package A is


```
...  
end A;  
  
-----  
  
with A;  
package B is  
...  
end B;
```

Section 10.3 states that B must be compiled after A. However, this implies that there is some well defined notion of compilation. Suppose I submit B to the compiler first; the compiler might carry out some processing, but delay some other processing until later, after A has been submitted; in fact, processing of B could be delayed until link time.

This difficulty is even more clearly seen in the context of recompilation. Suppose A is compiled first, and then B is compiled. Now, A is recompiled. Section 10.4 para 5 rules that B is now invalid, and must be recompiled (unless no longer needed). Here again, the need to process B depends on the compilation system, and on what "compilation" is defined to mean. For example, if A is totally unchanged when it is recompiled, then there is not need to recompile B; similarly, if at some earlier time B had been compiled against the new version of A, then a compilation system might retrieve the earlier results of compilation (now we would have to say that actually B has been recompiled BEFORE A was (most recently) recompiled, and this appears to break the letter of the rules in 10.3). Realization of these problems forces the acknowledgement in the last line of section 10.3 para 5, that an implementation might be able to reduce compilation costs by different behavior.

Either the LRM is deemed to define a notion of "compilation" which may not be appropriate for all situations, or an implementation must define "compilation" in an artificial way just so that the artificial rules in the LRM are obeyed. Here again the LRM is forced to complex lengths to take account of exceptional situations, because it has used an inappropriate method of description in the first place.

With respect to compilation failure, the question of whether the whole series of compilation units or just the one or ones with errors in them should be rejected, has been raised with the LRM before. This is simply not an appropriate question for the LRM, it should just be a property of the particular compilation system. That it is inappropriate is seen from the fact that a compiler could "cheat" by rejecting the whole compilation, but storing the results of successful units somewhere to be retrieved by the user if he wanted them.

The over specification of the compilation process causes a tight coupling between an Ada compiler and its library manager, and this makes it impossible to integrate an Ada compiler with an external configuration management system.

IMPORTANCE: ESSENTIAL

Without this change, developers of Ada compilers and environments will have to do considerable work to integrate compilation systems, and there are disincentives to original approaches to Ada development systems for fear that they will not conform to the letter of the reference manual. This will tend to restrict the availability of Ada compilers in association with sophisticated compilation management and configuration management systems, with a resulting restriction of choice to the user.

CURRENT WORKAROUNDS:

In general Ada compiler developers are forced to conform to the letter of the LRM. Where a more radical approach to compilation or configuration management is desired, the developer is forced into artificial devices or arguments to establish conformance, or the problem of conformance is quietly ignored.

Compilation systems are forced to restrict what they offer. For example, compilers at present generally cannot offer versions of units, which can make large projects difficult to manage, or they may offer concepts such as multiple libraries, which are of doubtful legality, but allow some degree of version management to be performed.

It is in general proving impossible to integrate compilers with APSEs, because the functionality of the compiler and the library management are not separated. Workarounds either involve modifications to the compiler to change the way they manage libraries or ad hoc partial solutions by extracting library information from the way it is held by and for the compiler and duplicating it in the environment.

POSSIBLE SOLUTIONS:

Simplify the definitions in chapter 10 of the LRM to omit procedural aspects such as those exemplified above. Rework the chapter into a pure declarative form. The explanations of possible consequences in terms of a procedural description could be given in the rationale (perhaps with some allusions to it in the notes in chapter 10).

A possible approach might include defining what is a complete and consistent set of units, mentioning the transitive closure of context clauses. A complete program should be defined statically in terms of the potential use of information.

There are a number of associated technical clarifications, including clarification of the relation between static and dynamic aspects of the program, clarification of the concept of a "main program" and the relation between it, tasks and the environment, and the change of the term "library unit" to "primary unit" for consistency with secondary unit, and to avoid the procedural connotations.

SUBUNIT NAMES

DATE: January 24, 1989

NAME: R. David Pogge

ADDRESS: Naval Weapons Center
EWTES - Code 6441
China Lake, CA 93555

TELEPHONE: (619) 939-3571
Autovon: 437-3571
E-mail: POGGE@NWC.ARPA

ANSI/MIL-STD-1815A REFERENCE: 10.1(3), 10.1(10), 10.2(5)

PROBLEM:

During the course of normal software development, large programs get broken down into body stubs which are written by different programming teams. Scope and visibility rules allow these teams to select the names of identifiers without worrying about the names of identifiers used by other programming teams. The names of subprograms, however, are unnaturally limited by chapter 10. Name clashes of subprograms are almost inevitable in a large program.

IMPORTANCE:

This correction would make the standard more consistent because it removes unnatural restrictions regulating the replacement of subprogram bodies with body stubs. It is upward compatible because it does not prohibit any currently legal Ada program from being compiled and linked. It may be easy for compiler vendors to implement because it removes one artificial restriction that must be checked.

This is certainly **IMPORTANT**, perhaps even **ESSENTIAL**.

If the revision is not accepted, it forces programmers to use the one of the workarounds described below.

CURRENT WORKAROUNDS:

There are two workarounds.

1. Compile a huge dictionary of all names used by project, force programmers to read and update it daily, and forbid them to use a subprogram name already in use.
2. Adopt a naming convention that guarantees unique simple names for all subprograms.

POSSIBLE SOLUTIONS:

Change the word **SIMPLE** to **EXPANDED** in the sentences of the paragraphs cited above.

This program compiles and runs on validated Ada compilers. The fact that the Put procedure is overloaded does not cause a problem.

```
with TEXT_IO;
procedure Simple_Name_Problem is

  procedure A is
    procedure Put is
    begin
      TEXT_IO.put_line ("This is line 1");
    end Put;
  begin
    Put;
  end A;

  procedure B is
    procedure Put is
    begin
      TEXT_IO.put_line("This is line 2");
    end Put;
  begin
    Put;
  end B;

begin
  A;
  B;
end Simple_Name_Problem;
```

If this procedure represented a large program, then procedures A and B would certainly be body stubs written by two separate programming teams. Both programming teams might accidentally use the same name for a subprogram. The program below raises an error at compile time on DEC Ada, and at link time on Alsys Ada.

```
with TEXT_IO;
procedure Simple_Name_Problem is

  procedure A is separate;

  procedure B is separate;

begin
  A;
  B;
end Simple_Name_Problem;

separate(Simple_Name_Problem)
procedure A is
  procedure Put is separate;
begin
  Put;
end A;

separate(Simple_Name_Problem)
procedure B is
```

```
        procedure Put is separate;
begin
    Put;
end B;

separate(Simple_Name_Problem.A)
procedure Put is
begin
    TEXT_IO.put_line("This is line 1");
end Put;

separate(Simple_Name_Problem.B)
procedure Put is
begin
    TEXT_IO.put_line("This is line 2");
end Put;
```

**GLOBAL NAME-SPACE CONTROL THROUGH MULTI-LEVEL
PROGRAM LIBRARIES
(ADA-UK/010)**

DATE: March 20, 1989

NAME: M.J. Pickett (from material supplied by members of Ada UK)

ADDRESS: Sema Group plc
Orion Court
Kenavon Drive
Reading, Berkshire
RG1 3DQ
UK

TELEPHONE: +44734508961

ANSI/MIL-STD 1815A REFERENCE: 10.1(3)

PROBLEM:

Within a program library the simple names of all library units must be distinct identifiers. To avoid name clashes in the program library during the development of large systems it becomes necessary to adopt unnatural naming conventions. The importation of "reusable" library units from more than one developer can result in name clashes which cannot be overcome.

IMPORTANCE: ESSENTIAL for large systems.

When developing large systems, it is usual to attempt to minimize the risk of having to undertake major recompilations as a result of some change. One approach to this is to avoid deep nesting of declarations, and instead, compile many library units. The requirement for uniqueness of name for these units is a management overhead which militates against the use of Ada.

The development of commercial sources of reusable software relies on the ability to distribute such software in other than source form. The possibility of being unable to import such software into a program library due to name clashes is a barrier to that development.

CURRENT WORKAROUNDS:

Where the software is entirely under the control of the application developer, unnatural naming conventions may be used to avoid name clashes.

Where a name clash occurs and source code is available, it may be possible to edit every reference to one of the library units to avoid the clash. This can result in the need for otherwise uncalled for recompilations and tests and may provide configuration management complications.

POSSIBLE SOLUTIONS:

The program library really requires a structure which enforces a restricted visibility of the names of library units. For many purposes a simple two level structure might suffice. Library units in the upper level would

be available for inclusion in any context clause, as now. Thus they are the "shareable" library units. Their simple names would be required to be unique within the set, as now. At the lower level, library units would form subsystems. Each subsystem would include one or more of the library units from the upper level. The names of library units would be unique within their subsystem and could be used in the context clauses of other library units in the same subsystem. Their context clauses could also refer to library units in the upper level, but no reference could be made to lower level library units in other subsystems. Thus the lower level library units would only be available to the higher level library units in the same subsystem.

The two level structure could be generalized to full tree structure.

Control over which level and subsystem form the destination for a compilation might be by pragma or some external means. It should be noted that some Ada compilation systems already provide this kind of facility for use during program development, but are forced by the standard to ensure uniqueness of library unit names before a program can be put together. Therefore the requirement is for a relaxation of the rules as expressed by the first and last sentences of 10.1(3).

CLEANUP AFTER MAIN SUBPROGRAM

DATE: October 25, 1988

NAME: S. Tucker Taft

ADDRESS: Intermetrics, Inc.
733 Concord Avenue
Cambridge, MA 02138

TELEPHONE: (617) 661-1840
E-mail: stt@inmet.inmet.com
E-mail: uunet!inmet!stt

ANSI/MIL-STD-1815A REFERENCE: 10.1(8)

PROBLEM:

Frequently when implementing large subsystems in Ada, there is a requirement for some kind of cleanup at the end of the program. Examples would be the releasing of locked resources, the flushing of buffers, the closing of files, etc. Currently, this program finalization is handled in a compiler-dependent way, typically involving attaching the address of a parameterless procedure to some list which is walked by the compiler's run-time system after completion of the main subprogram. This creates an undesirable compiler-dependence in code which otherwise has compiler-independent semantics.

IMPORTANCE:

Packages requiring finalization will necessarily be compiler-dependent.

CURRENT WORKAROUNDS:

The classic example is a subsystem which implements some kind of indexed-sequential access method. Such subsystems always perform as much buffering as possible, and frequently will delay writing out changed blocks if possible. However, at program exit it is essential that the buffer be flushed. This is currently possible in some compilers by adding a procedure address to a list maintained by the run-time system, as follows:

```
generic
  with procedure Cleanup;
package Cleanup_Action is end Cleanup_Action;

package body Cleanup_Action is
  -- create queue element containing address
  -- of Cleanup procedure, and put it on front
  -- of list so Cleanup procedures are executed
  -- in reverse order of elaboration
  ...
end Cleanup_Action;
...
with Cleanup_Action;
package body Isam_Subsystem is
```



```
...  
  procedure Flush_Buffers is ...;  
  package Flush_Cleanup_Action is new  
    Cleanup_Action(Flush_Buffers); -- Queue up finalization  
end Isam_Subsystem;
```

POSSIBLE SOLUTIONS:

Define a standard generic package, analogous to Cleanup_Action defined above, which provides for program finalization. This should involve no changes to the compiler, though it will imply a minor change to the run-time system.

DIFFICULTIES TO BE CONSIDERED:

The finalization actions have to be performed either before or after library-task termination, or perhaps interspersed based on library-package elaboration order. It is not immediately clear which order makes the most sense.

A more general package finalization mechanism would be preferable, allowing finalization code to be associated with any package, whether it is a library-level package or a nested package. However, this represents a more significant change to compilers, and is probably more appropriate for Ada 200X.

CONTEXT CLAUSES AND APPLY

DATE: March 22, 1989

NAME: E.N. Thomas

DISCLAIMER: The views expressed in this note are those of the author, and do not necessarily represent those of SD-Scicon PLC.

ADDRESS: SD-Scicon PLC
Pembroke House
Pembroke Broadway
CAMBERLEY
Surrey
UK
GU15 3XD

TELEPHONE: +44 276 686200

ANSI/MIL-STD-1815A REFERENCE: 10.1.1(3, 4), 10.5(4).

PROBLEM:

Section 10.1.1(4) explains that context clauses apply to the unit, its body (secondary unit), and its subunits, as relevant. Section 10.1.1(3) and 10.5(4), however, include cases where the context clause does not fully apply by prohibiting a use clause or a pragma ELABORATE mentioning names from context clauses that apply apart from the one of the current unit.

An extra issues concerning pragma ELABORATE is that it is not allowed to occur within a context clause when this contains more than one with clause and/or use clause.

IMPORTANCE: ESSENTIAL

Regularity is compromised without these, and the design goals of 1.3(3) are violated ("a small number of underlying concepts integrated in a consistent and systematic way").

CURRENT WORKAROUNDS:

It is necessary to repeat unit names from a parent's context clause in order to include a use clause or pragma ELABORATE in a body or sub-unit, even though the context clause applies here.

POSSIBLE SOLUTIONS:

Extend the concept of the context clause "applying" to include the ability to mention unit names in use clauses and as arguments to pragma ELABORATE when the unit names are in any context clause that applies to the unit.

In order to allow this when the body of sub-unit needs no further with clauses, it is necessary to extend the syntax in 10.1.1, since at present a with clause must precede a use clause. In doing this, the exact position for pragma ELABORATE has been indicated -- in fact this ought to be done for all predefined pragmas. Normal restrictions like using only names that have already appeared in a with clause would remain.

```
context_clause ::= {context_item}  
context_item ::= with_clause | use_clause | elaborate_clause  
elaborate_clause ::=  
  pragma ELABORATE  
    (library_unit_simple_name {, library_unit_simple_name}_);
```

This then allows context clauses like for following.

```
with TEXT_IO;  
package A is  
...  
end A;  
  
use TEXT_IO;  
package body A is  
...  
end A;  
  
with A;  
use A;  
pragma ELABORATE (A);  
with TEXT_IO;  
package B is  
...  
end B;
```

ALLOW SEMICOLON AFTER SEPARATE CLAUSE

DATE: January 15, 1989

NAME: William Thomas Wolfe

ADDRESS: Home: 102 Edgewood Avenue #2
Clemson, SC 29631 USA

Office: Department of Computer Science
Clemson University
Clemson, SC 29634 USA

TELEPHONE: Home: (803) 654-7030
Office: (803) 656-2847
E-mail: wtwolfe@hubcap.clemson.edu

ANSI/MIL-STD-1815A REFERENCE: 10.2(2)

PROBLEM:

A subunit is defined such that one cannot have a semicolon after the right parenthesis which encloses the parent_unit_name.

CONSEQUENCES:

Since subunits generally appear with the following indentation, separate (parent_unit_name) proper_body the separate clause strongly resembles the statement-like with clauses and use clauses, and there is a strong programmer tendency to append a semicolon to anything that resembles a statement. The fact that separate clauses violate this general rule that anything resembling a statement ends with a semicolon leads to syntax errors and to considerable cognitive dissonance in the mind of the programmer.

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Redefine subunit to include at least the option of having a semicolon after the right parenthesis which encloses the parent_unit_name.

ALLOW SUBUNITS WITH SAME ANCESTOR LIBRARY**DATE:** February 17, 1989**NAME:** Arny B. Engelson**ADDRESS:** AT&T Bell Laboratories
Whippany Road (14B-257)
Whippany, NJ 07981**TELEPHONE:** (201) 386-4816
E-mail: att!wayback!arny**ANSI/MIL-STD-1815A REFERENCE:** 10.2(5)**PROBLEM:**

Subunits that have the same ancestor library unit must have distinct identifiers. This eliminates the ability to make subunits out of overloaded subprograms. In Ada, where overloading is a very commonly used feature, this is an inconsistent (seemingly arbitrary) restriction. Subunits that have the same ancestor library unit should be allowed so long as they have different parameter and result type profiles (ARM 6.6).

I have many library packages of subprograms with the same name, but with different parameters. Not being able to make these into subunits causes me to have much larger source files, reduces the modularity of my program, and causes unnecessary recompilation when I make a change.

IMPORTANCE:

An individual's scale of 1 to 10 tends to be very subjective. Therefore, I will only say the problem is of fairly major importance. It places unfair restrictions of the programmer, and is not in the spirit of Ada.

CURRENT WORKAROUNDS:

Only one in a set of overloaded subprograms can be a subunit, or the structure of the program much be changed.

POSSIBLE SOLUTIONS:

Subunits that have the same ancestor library unit should be allowed so long as they have different parameter and result type profiles. In other words, overload resolution should take place according to ARM 6.6.

ADDITIONAL COMMENTS:

Arbitrary language restrictions to make things easier for compiler implementors should not be present in a mature language.

REDUCING COMPILATION COSTS

DATE: July 13, 1989

NAME: M. Ben-Ari

ADDRESS: Brandeis University (until 8/89)

81 Hagedud Haivri St. (from 9/89)
Kiryat Haim 26306
Israel

TELEPHONE: 617-736-2726 or 617-332-1419 (until 8/89)
011-972-4-725905 (from 9/89)
E-mail: moti@cs.brandeis.edu (until 8/89)

ANSI/MIL-STD-1815A REFERENCE: 10.3.5

PROBLEM:

Laboratory and field testing of embedded computer systems requires constant modification of the software being tested, for example to temporarily ignore a spurious input or a bug that cannot be immediately fixed. The large semantic gap between Ada and the machine code makes "patching" the object code extremely difficult. Testing is often done in an environment that may not have access to a full software development facility. Thus it is important to minimize re-compilation requirements so that a small change can be done with a host of limited resources.

The standard apparently intended that compilers attempt to minimize recompilation, though this is not done in the compilers that I have used:

"An implementation may be able to reduce the compilation costs if it can deduce that some of the potentially affected units are not actually affected by the change." (LRM 10.3)

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

The current workaround is to use compilation time as a DESIGN requirement which is surely not consistent with modern software engineering practice. The problem is particularly acute with respect to subunits since they all must be compiled when the parent unit is modified even if it is clear that re-compilation is unnecessary.

POSSIBLE SOLUTIONS:

A subunit will NOT be re-compiled if any of the following is changed in the parent unit:

- (i) Comments.
- (ii) Initial values of objects declared in the parent unit.
- (iii) Bodies of procedures, packages or tasks declared in the parent unit.

Example:

A real-time controller may need to have extensive calibration done to scaling tables and equations long after the interfacing has been completed. It is natural to separate out debugged entities as subunits and continue to modify the main program. Current compilers require the entire program to be re-compiled.

```
procedure Main is
  Table: Table_Type := ... ;

  task Background;
  task body Background is separate;
  procedure Scale_Input is separate;
  procedure Scale_Output is separate;

  procedure Compute is ...;

begin
  loop
    Scale_Input;
    Compute;
    Scale_Output;
  end loop;
end Main;
```

TRANSITIVE PRAGMA ELABORATE

DATE: October 25, 1988

NAME: S. Tucker Taft

ADDRESS: Intermetrics, Inc.
733 Concord Avenue
Cambridge, MA 02138

TELEPHONE: (617) 661-1840
E-mail: stt@inmet.inmet.com
E-mail: uunet!inmet!stt

ANSI/MIL-STD-1815A REFERENCE: 10.5(3)

PROBLEM:

The current semantics of pragma Elaborate are unfortunate since no transitivity is implied. If subprograms of a package are called at library-unit elaboration time, then the caller must "pragma-Elaborate" that package. However, if the subprograms of that package call subprograms of other packages, then an elaboration-order `PROGRAM_ERROR` is still possible, unless the outermost calling package pragma-Elaborates all such lower-level packages as well. This requires an undesirable amount of knowledge about the inner-workings of the pragma-elaborated package, resulting in excess dependencies.

Ideally, a pragma `Transitive_Elaborate` would apply pragma Elaborate to all packages "withed" by the target package, directly or indirectly, unless there is an explicit direct pragma Elaborate which specifies an opposite order. This would significantly reduce the number of pragma elaborates required in a complex system that includes elaboration-time computation.

IMPORTANCE:

Additional pragma elaborates with attendant additional dependencies will be required, and the possibility for circularities is increased. Notice that `Transitive_Elaborate` does **not** specify the order of elaboration of the lower-level packages relative to the target package, but rather simply requires that they all be elaborated before the calling package.

CURRENT WORKAROUNDS:

A simple example can be drawn from the ACVC suite. Originally, tests which used package Report at elaboration time only "pragma-elaborated" package Report. However, later it was determined that several compilers raised `PROGRAM_ERROR` because package Report did not itself pragma-elaborate `Text_IO` (though it did "with" it).

A pragma like `Transitive_Elaborate` would solve such problems, and many others which come up in complex systems.

POSSIBLE SOLUTIONS:

Provide a pragma like Transitive_Elaborate (or call it simply Elaborate if it can be shown to be upward compatible), which ensures that the target package and all packages it "withs" directly or indirectly are elaborated prior to the package containing the pragma.

For additional references to Section 10. of ANSI/MIL-STD-1815A, see the following sections, revision request numbers, and revision request titles in this document.

REVISION REQUEST

<u>NUMBER</u>	<u>TITLE</u>	<u>SECTION</u>
0117	PRE-ELABORATION	3
0092	FINALIZATION	3

**ADA 9X REVISION REQUESTS
THAT REFERENCE
ANSI/MIL-STD-1815A**

SECTION 11. EXCEPTIONS

PASS EXCEPTIONS AS PARAMETERS OR ACCESS EXCEPTION'S 'IMAGE

DATE: January 15, 1989

NAME: William Thomas Wolfe

ADDRESS: Home: 102 Edgewood Avenue #2
Clemson, SC 29631 USA

Office: Department of Computer Science
Clemson University
Clemson, SC 29634 USA

TELEPHONE: Home: (803) 654-7030
Office: (803) 656-2847
E-mail: wtwolfe@hubcap.clemson.edu

ANSI/MIL-STD-1815A REFERENCE: 11.

PROBLEM:

Cannot pass exceptions as parameters, or access an exception's 'IMAGE.

CONSEQUENCES:

It is not possible to program a procedure ASSERT which takes as parameters a Boolean expression and an exception, and raises the desired exception if the Boolean expression turns out to be false. It is also not possible to generate warning messages to the effect that a particular exception was handled by the program if the name of the exception is not known in advance (e.g., in an others clause).

CURRENT WORKAROUNDS:

The ASSERT procedure can raise Assertion_Failed, and the call to the ASSERT procedure can be encapsulated in a local block which raises the desired exception upon encountering Assertion_Failed. However, this is unnecessarily verbose and tends to defeat the purpose of having an ASSERT procedure, since half of the logic involved in processing the assertion must appear outside the procedure call.

There is no workaround to the problem of finding out the image of an exception, in the general case. One could sit there and type when Some_Very_Long_And_Descriptive_Exception

=> PUT ("Some_Very_Long_And_Descriptive_Exception was handled..."); for every assertion whose name could be determined in advance, with obvious penalties in terms of code readability and cost.

POSSIBLE SOLUTIONS:

Revise ANSI/MIL-STD-1815A such that exceptions can be passed as parameters and such that exceptions have a 'IMAGE attribute.

FINDING THE NAME OF THE CURRENTLY RAISED EXCEPTION**DATE:** April 7, 1989**NAME:** Tony Orme**ADDRESS:** ATM Computer
AEG (UK) Ltd.
Engineering Division
Eskdale Road
Innersh
Wokingham
Berkshire
RG11 5PF
UK**TELEPHONE:**

Country Code	44
Direct line	734 441419
Via Switchboard	734 698330 ex124
FAX	734 441397

ANSI/MIL-STD-1815A REFERENCE: 11.**PROBLEM:**

In several common situations, it is important to be able to know the name of the currently raised exception, even when outside its scope.

These common situations are:

1. In on-line (not real-time) systems, where a human operator may have a choice of actions (say Retry, Abort or Detour) based on the exception raised;
2. Test harnesses and similar software should be able to print exceptions raised, perhaps with time, task id and other pertinent data, in reports;
3. Anywhere that anonymous exceptions may be handled. For example, in using a commercial package where some spec (it may be objected that a commercial company should not supply such packages, but it is unwise overly to rely on a "shouldn't"). For tracing and other purposes, it is necessary to know what the exception was and where it was raised.

The problem is actually a composite of two slightly different problems:

- a. In 1 and 2 above, the range of exceptions may be known, but an easy way displaying the current one is required (i.e., a case statement is cumbersome and awkward to maintain in this context);
- b. In 3 above, an unknown exception may be raised, over which there is no control. There has to be a way of discovering its identity.

IMPORTANCE: IMPORTANT

Effect on Current Applications: Possible solutions (see below) will have no impact on current applications.

Consequences of Non-implementation: If this amendment is not actioned, the penetration of Ada outside the military/aerospace sector may be impeded.

CURRENT WORKAROUNDS:

The D.G./Rolm ADE Ada compiler provides a non-standard package called `CURRENT_EXCEPTION`, with a single function `NAME` returning a string. (If there is no currently raised exception, a null string is returned.) However, in D.G./Rolm sites, this feature is frequently not used because it is completely non-portable.

POSSIBLE SOLUTIONS:

Because the raised exception may be out of scope, a means of returning a string rather than an exception is required.

The question of redeclared exceptions should also be considered. While it may be possible, except in unnamed blocks, to provide the name of the place where the exception was declared, it is felt on balance to be unnecessary.

Four possible solutions are:

1. A function to return the required information might be provided as a standard library unit. This has the merit of being relatively straightforward to implement.
2. A task attribute might be defined such as `T_EXCEPTION`. An extension would have to be made to permit enquiry on the main program. While this form allows wider possible use than solution 1, and is eminently suitable for situation 2 above, it is probably unnecessary complex.
3. The existing attribute `IMAGE` might be coupled with a predefined enumeration type, e.g., `EXCEPTION_RAISED_IMAGE`. This is really very similar to solution 1.
4. A string could be declared in the form of an (optional) parameter of an exception handler, e.g.,

```
exception (S : string)
when
```

There would be no corresponding actual parameter as exception handlers are not explicitly called. The situation where no exception has been raised does not arise. This may be close to an elegant solution.

IMPLEMENTATION OF EXCEPTIONS AS TYPES

DATE: May 30, 1989

NAME: David Papay

ADDRESS: GTE Government Systems Corp
PO Box 7188 M/S 5G09
Mountain View, CA 94039

TELEPHONE: (415) 694-1522
E-mail: papayd@gtewd.af.mil

ANSI/MIL-STD-1815A REFERENCE: 11., 3., 6., 7., 8., 12.

PROBLEM:

Currently, exceptions are distinct entities. There is no way to "group" related exceptions and handle them collectively, nor is there a way to declare a general "class" of exceptions and declare individual exceptions of that class. Finally, there is no way to pass exceptions as parameters to generic units, or to subprograms.

IMPORTANCE: IMPORTANT.

CURRENT WORKAROUNDS:

The current language rules allow multiple exceptions to be handled by one handler using exception choices, so this does solve part of the problem.

POSSIBLE SOLUTIONS:

Implement exceptions as types (in specific, as limited private types). Certain parts of the Ada syntax would change as follows:

```
3.3.1(2) type_definition ::=
    ...
    | exception_type_definition
```

```
11.1 (2) (?) exception_type_definition ::= exception;
```

```
11.2 (2) exception_choice ::=
    ...
    | in subtype_indication
```

where the base type of the subtype indication is an exception type.

Thus, the current syntax for declaring exception would still be legal:

```
QUEUE_OVERFLOW : exception;
```

```
QUEUE_UNDERFLOW : exception;
```

These declarations would declare two exception objects of anonymous exception types. They would be treated as objects of a limited private type (this is very similar to task objects of anonymous task types). However, it would now also be possible to declare the following:

```
type STACK_ERROR is exception;
```

```
STACK_OVERFLOW : STACK_ERROR;
STACK_UNDERFLOW : STACK_ERROR;
```

Exception objects are constants, but the reserved word "constant" does not appear in an exception object declaration. Since exception objects are limited private, assignment is not defined for them, and there is no explicit initialization (cf task objects, 9.2(6)).

Exception handlers could be written:

```
exception
when in STACK_ERROR =>
  -- This would handle both STACK_OVERFLOW and STACK_UNDERFLOW
```

But, of course, handlers could still be written:

```
exception
when STACK_OVERFLOW =>
  ...
when STACK_UNDERFLOW =>
  ...
```

Subtypes of exception types can be declared, however, no constraints are applicable (cf. task types, 9.2(8)). Renaming declarations cannot be used to rename exception types, but can be used to rename individual exception objects of exception types. The current form of exception renaming would be used to rename anonymous exceptions.

```
subtype NEW_EXCEPTIONS is STACK_ERROR;
```

```
OVERFLOW : STACK_ERROR renames STACK_OVERFLOW;
```

```
UNDERFLOW : exception renames QUEUE_UNDERFLOW;
```

Some other details to be worked out include:

- 1) Is the following exception handler legal? What would its effect be?

```
exception
when STACK_UNDERFLOW =>
  ...
when in STACK_ERROR =>
  ...
```


- 2) What are the allowed modes of task types as subprogram parameters, and generic formal objects?
- 3) Can task types be generic formal types? Would there be a special form of generic type definition, or would a limited private type definition be used? I would suggest a new generic type definition:

12.1(2)

```
generic_type_definition ::=  
    ...  
    | exception_type_definition  -- defined previously in this request.
```

- 4.) Should the predefined exceptions remain objects of anonymous type, or should there be two predefined exception types

-- in STANDARD:

```
type PREDEFINED_EXCEPTION is exception;  
  
CONSTRAINT_ERROR,  
NUMERIC_ERROR,  
...  
TASKING_ERROR    : PREDEFINED_EXCEPTION;
```

-- in IO_EXCEPTIONS:

```
type IO_EXCEPTION is exception;  
  
STATUS_ERROR,  
MODE_ERROR,  
...  
LAYOUT_ERROR    : IO_EXCEPTION;
```

NOTE:

This request is similar to one submitted by Damon Lease (also of GTE Government Systems at that time). I have expanded on the original idea somewhat, and have addressed some issues he and I had not originally thought of.

PROVIDE USER-SPECIFIED STORAGE RESERVE FOR RECOVERY FROM STORAGE_ERROR

DATE: June 7, 1989

NAME: Dan Stock (ACM Special Interest Group on Ada, Ada Runtime Environment Working Group)

ADDRESS: R.R. Software, Inc.
2145 Crooks Rd. #50
Troy, MI 48084

TELEPHONE: E-mail: stockd@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 11.

PROBLEM:

Applications should be able to specify that some "stack" storage be held in reserve for use in recovery from `STORAGE_ERROR`.

SPECIFIC REQUIREMENT/SOLUTION CRITERIA:

Ada should provide a means for an application to reserve a dynamic amount of storage for recovery from `STORAGE_ERROR`. The application must be able to utilize the reserve when the need arises. When `STORAGE_ERROR` is raised because the available storage for items commonly allocated from the "stack" is exhausted, it is likely that there is insufficient storage for an exception handler to execute in order to meaningfully recover from the exception. As a consequence, a second `STORAGE_ERROR` may be raised in the attempt to handle the earlier `STORAGE_ERROR`, and the recovery operation is abandoned. It should be possible for an application to guarantee that the recovery action can be performed.

JUSTIFICATION:

Since some software systems cannot be permitted to fail, it is often unacceptable for a program to terminate, or for a task to complete, due to `STORAGE_ERROR` being raised. Highly stochastic applications cannot afford to size memory for the worst case state of the system, so static preallocation is not workable. Therefore an application must be able to recover from exhaustion of storage. The application may require additional storage to perform needed recovery actions; the amount of storage needed is application-specific and potentially dynamic.

The current standard allows the run-time environment to consume all available storage before raising `STORAGE_ERROR`. There are no means for an application to define the storage it needs for recovery. An all-too-common consequence of this is that a task may not be able to do anything but silently complete once `STORAGE_ERROR` has been raised.

The situation is particularly difficult for long-running, real-time, programs. For Ada to support such applications, it must be possible to fully recover from a failure of the memory management imposed by Ada's run-time environment.

DIFFICULTIES:

Possible solutions are likely to be implementation-dependent, due to the varying ways that Ada run-time environments use memory. For example, a solution that is useful in many uniprocessor environments may not be meaningful in a distributed system.

POSSIBLE SOLUTIONS:

One possible solution is for the language to define a representation clause to specify the storage to be reserved for recovery from any `STORAGE_ERROR` handled in a program unit. In an environment where activation records for the execution of subprogram calls are placed on a run-time stack, the reserved storage could be placed at the end of the activation record for the subprogram call. The reserved storage could be implicitly made available if and when the appropriate `STORAGE_ERROR` handler is entered.

RETRIEVE CURRENT EXCEPTION NAME

DATE: July 17, 1989

NAME: Mitch Gart

ADDRESS: Alsys Inc.
67 South Bedford Street
Burlington, Mass 01803

TELEPHONE: (617) 270-0030
E-mail: Brosgol@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 11.

PROBLEM:

There should be a way to print the name of an exception from within an "others" handler. This request is in behalf of the IEEE 1003.5 committee doing the POSIX-Ada binding. We have a requirement to be able to print the current exception name, because we declare many exceptions in the POSIX-Ada binding and it would be inconvenient for an application to enumerate all of them in an "others" handler just to print an error message.

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS:

Applications must write code that enumerates all possible exceptions, or must rely on a non-portable solution provided by an Ada compiler vendor.

POSSIBLE SOLUTIONS:

We wanted to adopt a solution that would be the same as the Ada 9X solution to this problem, and not invent a POSIX-specific solution. We chose the following solution because somebody on the committee thinks it is the same as one that was already submitted to Ada 9X. If so, then this request is another vote to add the functionality.

```
package CURRENT_EXCEPTION is
  -- return the simple name of the last exception raised in the
  -- current task;
  function EXCEPTION_NAME return STRING;
end;
```

This functionality is:

- guaranteed to work in multi-tasking programs (EXCEPTION_NAME returns the name of the last exception raised in the current task);
- only guaranteed from within a handler (otherwise returns empty string);
- the name returned is a simple name (for example, "CONSTRAINT_ERROR", not "STANDARD.CONSTRAINT_ERROR").

HANDLING OF UNSUCCESSFUL ATTEMPTS TO ALLOCATE MEMORY

DATE: June 7, 1989

NAME: Offer Pazy (ACM Special Interest Group on Ada, Ada Runtime Environment Working Group)

ADDRESS: Software Leverage Inc.
485 Massachusetts Ave.
Arlington, MA 01274

TELEPHONE NUMBER: (617) 648-1414
E-mail: offer@sli.com
E-mail: pazyo@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 11.1

PROBLEM:

The memory in an embedded, real-time application can be highly dynamic; it is often repeatedly allocated and de-allocated. As a result, an attempted allocation may not succeed immediately because memory is temporarily exhausted. Yet, if the allocation were attempted a short time later, it would succeed.

Ada should provide more flexibility by permitting an application to specify an alternative to raising `STORAGE_ERROR` when there is insufficient storage to create an object via an allocator.

Specific Requirements/Solution Criteria:

The programmer should be able to specify alternatives to raising `STORAGE_ERROR` when an attempted allocation of a new object fails. At a minimum, the alternative actions should include waiting for the storage to become available.

CURRENT WORKAROUNDS:

For the class of applications in which worst case storage requirements exceed available storage, it is not feasible to use static storage for all of the application's critical storage.

With Ada as it is currently defined, an allocation failure always raises `STORAGE_ERROR`. Handling this error is an expensive operation in terms of execution time. Furthermore, in most cases, the application has little choice but to wait until the memory becomes available. Thus, each allocation must be surrounded by a loop that handles the `STORAGE_ERROR`, delays for a brief period, and then attempts the allocation again. This type of polling is very wasteful of computer time.

An alternative workaround would be to write a "roll your own" storage allocation package. However, this workaround is not acceptable for several reasons including (but not limited to) its potential lack of portability and inadequate performance.

Ada already provides a mechanism to queue requests for dynamic resources that may be temporarily unavailable. These resources are task entries. The same flexibility should be extended to memory resources. Not having alternatives for allocators is as limiting as always raising an exception whenever a rendezvous

partner is busy.

IMPORTANCE:

Tasks will have to poll until memory becomes available. This can lead to a significant increase in execution time that might hinder mission-critical applications.

POSSIBLE SOLUTIONS:

In addition to the minimum alternative cited above (i.e. waiting for the storage to become available), other desirable alternatives would include: (1) waiting up to a specified maximum time for the storage; and (2) returning a null pointer without raising any exception

EXCEPTION DECLARATIONS NOT SHARABLE**DATE:** October 28, 1988**NAME:** S. Tucker Taft**ADDRESS:** Intermetrics, Inc.
733 Concord Avenue
Cambridge, MA 02138**TELEPHONE:** (617) 661-1840
E-mail: stt@inmet.inmet.com
E-mail: uunet!inmet!stt**ANSI/MIL-STD-1815A REFERENCE:** 11.1(3)**PROBLEM:**

Because exceptions are defined "at compile time," exceptions are perniciously difficult to implement when attempting to share code across generic instantiations. Because of this, implementations of shared generics will be more complex, less efficient, and less common.

As specified in the LRM, each instantiation of a generic must have its own set of exceptions. However, each elaboration of a package must *not* create a new set of exceptions. These two requirements seem inconsistent, and mean that there is no straightforward way to implement the declaration of an exception in a generic package when instantiations are going to share code.

IMPORTANCE:

If exceptions remain the way they are, implementations attempting to share code between generic instantiations will remain complex, inefficient, and rare.

Each instantiation will have to implicitly declare all of the exceptions which are found within the generic being instantiated, wherever they appear (including within bodies, nested instantiations, subunits, etc.). This creates undesirable dependencies on the body of the generic, and requires that a table of exception descriptors be passed from the outermost instantiator to its instantiation and all nested instantiations/subunits.

CURRENT WORKAROUNDS:

The statement that "the particular exception denoted by an exception is determined at compilation time and is the same regardless of how many times the exception declaration is elaborated" is a unique situation in the language, and seems unjustified. Either exceptions should be totally statically defined, so that all instantiations share the same exceptions, or totally dynamically defined, so that a new exception is defined by each elaboration.

Given the current situation, the change which would involve the least interference with existing programs would be to make exceptions totally dynamic (i.e. each elaboration declares a new exception). The only way a program could detect that exceptions are dynamic would be by declaring an exception nested in a recursive program, and then attempting to handle an exception raised by a recursive call in an outer call. The normal way to do this would be to declare the exception at least one level out from the subprogram raising the

exception, so that it could be handled by its caller, in which case the change would be upward compatible.

POSSIBLE SOLUTIONS:

Make each elaboration of an exception declaration create a new exception. This will simplify sharing code between generic instantiations by allowing shared code to always create a new exception at each execution, rather than having to create / find the associated exception determined by the instantiator (or the instantiator's instantiator, or ...). It will also unify the semantics of exceptions with all other declarations within the language.

"Creating a new exception" can be as simple as storing the address of a string into a word in the local stack frame. The address of this word in the stack frame would then be the unique identifier for the exception.

Since most exceptions would be declared at the library level and would be as "static" as before, most existing code would not be adversely affected in performance.

"WHEN" CLAUSE TO RAISE EXCEPTIONS**DATE:** May 23, 1989**NAME:** Jeffrey R. Carter**ADDRESS:** Martin Marietta Astronautics Group
MS L0330
P.O. Box 179
Denver, CO 80201**TELEPHONE:** (303) 971-4850
(303) 971-6817**ANSI/MIL-STD-1815A REFERENCE:** 11.3(2)**PROBLEM:**

Unlike the "exit" statement [5.7(2)], an "if" statement must be used to conditionally raise an exception.

IMPORTANCE: ADMINISTRATIVE**CURRENT WORKAROUNDS:**

Use an "if" statement.

POSSIBLE SOLUTIONS:

Allow a "when" clause on "raise" statements similar to that for "exit" statements:

raise [exception_name] [when condition];

This is more readable than using an "if" because it is a more natural way of expressing the desired result. It would be fully compatible with the current standard.

INCLUDE "WHEN" IN RAISE STATEMENT SYNTAX**DATE:** July 9, 1989**NAME:** William Thomas Wolfe

ADDRESS: E-mail: wtwolfe@hubcap.clemson.edu
Home: 102 Edgewood Avenue #2
Clemson, SC 29631 USA

Office: Department of Computer Science
Clemson University
Clemson, SC 29634 USA

TELEPHONE: Home: (803) 654-7030
Office: (803) 656-2847

ANSI/MIL-STD-1815A REFERENCE: 11.3.2, 5.7.2**PROBLEM:**

The raise statement is quite similar to the exit statement in that they both serve to transfer the flow of control, almost always on the basis of some condition. The exit statement has a syntax which is appropriate for the purpose:

```
exit_statement ::= exit [loop_name] [when condition];
```

However, the raise statement is incomplete in this respect:

```
raise_statement ::= raise [exception_name];
```

CONSEQUENCES:

The incompleteness of the raise statement's syntax hampers the readability of Ada software, and causes some dissonance in the minds of Ada users in that an intuitively "natural" statement (raise Exception when Condition) is not permitted.

WORKAROUNDS: None**POSSIBLE SOLUTIONS:**

Revise 11.3.2 to read:

```
raise_statement ::= raise [exception_name] [when condition];
```

For additional references to Section 11. of ANSI/MIL-STD-1815A, see the following sections, revision request numbers, and revision request titles in this document.

REVISION REQUEST

<u>NUMBER</u>	<u>TITLE</u>	<u>SECTION</u>
0036	MAKE EXCEPTION A PREDEFINED TYPE	3
0111	FAULT TOLERANCE	5

**ADA 9X REVISION REQUESTS
THAT REFERENCE
ANSI/MIL-STD-1815A**

SECTION 12. GENERIC UNITS

ALLOW OVERLOADING OF GENERIC PARAMETER STRUCTURES

DATE: January 15, 1989

NAME: William Thomas Wolfe

ADDRESS: Home: 102 Edgewood Avenue #2
Clemson, SC 29631 USA

Office: Department of Computer Science
Clemson University
Clemson, SC 29634 USA

TELEPHONE: Home: (803) 654-7030
Office: (803) 656-2847
E-mail: wtwolfe@hubcap.clemson.edu

ANSI/MIL-STD-1815A REFERENCE: 12.

PROBLEM:

It is not possible to overload generic packages with different generic parameter structures.

CONSEQUENCES:

Suppose that I have a generic linked list package; sometimes I need to be able to INPUT and OUTPUT the list to and from a file, and sometimes I do not need this capability and therefore do not wish to provide INPUT and OUTPUT procedures for the abstract data type over which the linked list is instantiated. A natural solution would be to overload the generic package name `GENERIC_LINKED_LIST` with two different generic parameter structures, one which includes the INPUT and OUTPUT procedures and another which does not. Naturally, the package which does not require INPUT and OUTPUT as generic parameters will also not provide INPUT and OUTPUT for the linked list type. However, this is presently impossible. I don't believe ANSI/MIL-STD-1815A specifically addresses the issue, but at least some validated compilers do not accept overloadings of generic parameter structures.

CURRENT WORKAROUNDS:

One can use a different name such as `GENERIC_LINKED_LIST_WITH_IO`, but the number of different options could easily exceed one, thus resulting in incredibly long package names. Such extremely long package names arise frequently, as evidenced by the contents of the book "Software Components with Ada", by Grady Booch.

POSSIBLE SOLUTIONS:

Revise ANSI/MIL-STD-1815A such that the overloading of generic parameter structures is specifically permitted.

UNCONSTRAINED SUBTYPES AS GENERIC ACTUALS

DATE: October 28, 1988

NAME: S. Tucker Taft

ADDRESS: Intermetrics, Inc.
733 Concord Avenue
Cambridge, MA 02138

TELEPHONE: (617) 661-1840
E-mail: stt@inmet.inmet.com
E-mail: uunet!inmet!stt

ANSI/MIL-STD-1815A REFERENCE: 12.3.2(4)

PROBLEM:

There should be some way to explicitly allow or disallow actual generic types being unconstrained in the generic spec.

A proposed syntax would be:

<code>type A(<>) is private;</code>	-- Unconstrained allowed
<code>type A() is private;</code>	-- Unconstrained disallowed

This would improve the "contract" specified by the generic spec, and eliminate the current bizarre checking required when a generic body is compiled which may require verifying that all existing instantiations do not specify an unconstrained subtype as an actual. This bizarre checking not only affects the compiler, but also can result in unexpected and sometimes inexplicable errors for the user to handle.

IMPORTANCE:

Generics having formal private types will be less dependably reusable if there is no specification in the spec whether unconstrained actuals are allowed.

In systems supporting shared sub-libraries, it is generally impractical to locate all instantiations at the time of recompilation of a generic body, and hence the unsuspecting instantiators will get burned when they update to the newer release of the sub-library.

The definer of a generic will be given no help at compile-time for finding the illegal uses of the formal type which ultimately preclude the instantiation with an unconstrained actual.

CURRENT WORKAROUNDS:

As the language is now defined, there is no way to tell from a generic spec whether the actual subtype for a given formal type may be unconstrained. This creates undesirable dependencies on the body and/or particular instantiations, and makes sharing code between instantiations that much more difficult, since generally the treatment of unconstrained types and constrained types is very different at the generated code level.

POSSIBLE SOLUTIONS:

As suggested above, a proposed syntax would be:

<code>type A(<>) is private;</code>	-- Unconstrained allowed
<code>type A() is private;</code>	-- Unconstrained disallowed (unless has
	-- discriminant defaults)

Another possible syntax is to allow a representation-like clause -- "for A'constrained use False" or "for A'constrained use True" for allowed/disallowed, though it is a little misleading since the real rule does not disallow unconstrained types so long as there are defaults for discriminants, and unconstrained is allowed, it is not actually required (presumably).

In any case, if unconstrained is allowed, then at compile time of the generic spec and body, use of the formal type would be illegal in circumstances requiring a constrained type (or defaults for the discriminants).

If unconstrained is disallowed, then at compile time of the instantiation, it would be illegal to specify an unconstrained subtype (unless the discriminants had defaults).

This solution (or equivalent) would remove undesirable dependencies between generic bodies and their instantiations, and would simplify legality checking in the compiler at both instantiation time and body recompilation time, since there would be no need to search the library for other units which might be made illegal by the current unit.

Upward compatibility concerns mean that the "non-committal" syntax "type A is private" should remain supported, though probably "denigrated" and intended for removal in Ada 200x.

For additional references to Section 12. of ANSI/MIL-STD-1815A, see the following sections, revision request numbers, and revision request titles in this document.

REVISION REQUEST

<u>NUMBER</u>	<u>TITLE</u>	<u>SECTION</u>
0117	PRE-ELABORATION	3
0101	IMPLEMENTATION OF EXCEPTIONS AS TYPES	11

**ADA 9X REVISION REQUESTS
THAT REFERENCE
ANSI/MIL-STD-1815A**

**SECTION 13. REPRESENTATION CLAUSES AND
IMPLEMENTATION-DEPENDENT FEATURES**

BUILDING OBJECT PROGRAMS

DATE: February 10, 1983

NAME: Bill Taylor (from material supplied by members of Ada-UK)

ADDRESS: Ferranti Computer Systems Limited
Ty Coch Way
Cwmbran
Gwent
NP44 7XX
UK

TELEPHONE: +44 6333 71111

ANSI/MIL-STD-1815A REFERENCE: 13.

PROBLEM:

Ada currently supports the use of various features in the source code to control the process of building the object program, examples being length clauses to control task stack sizes and collection sizes.

It is inconvenient from a project control viewpoint, and often inefficient from a recompilation viewpoint, for changes to this information to be treated in the same manner as changes to the (rest of the) source code.

Additionally, there are other language features, such as pragmas, for which the same problems arise. In either case, the information controls the compilation or building process rather than the logic of the source code.

When Ada units, containing control information are under development (including unit testing), the information is either unavailable or subject to change during integration. It is also likely that, if the unit is reused in another system, then it will be precisely this information that is likely to differ. It is most important that reuse does not necessitate maintaining multiple copies. That is, control information should not be treated as physically part of the source. In some cases, changing such information should obsolete the unit and necessitate a recompilation. In all cases, a rebuild will be necessary.

IMPORTANCE: IMPORTANT

Without a solution to this problem, projects will be harder to control, compilation resources will be wasted, and the benefits of reuse will only be partially realized.

CURRENT WORKAROUNDS:

If any workaround is provided by the vendor, it is by the use of a vendor-specific build control language. There is little incentive to define a standard Build Control Language.

POSSIBLE SOLUTIONS:

The most obvious solution is to physically separate the control information from the rest of the program text. Ideally, a secondary standard would be defined as to how such information should be provided. Such

a standard should permit extensions, although it would be highly desirable for such extensions to evolve in a uniform way.

For reasons of compatibility with the current standard, the new standard would have to allow the currently defined control information to be provided interleaved with declarations.

A distinction can be made between information that controls the compilation process and that which controls only the build process, although the distinction may be implementation dependent.

The pragmas `Memory_Size`, `Storage_Unit` and `System_Name`, are intended for initializing the program library.

The pragmas `Priority` and `Elaborate`, the length clause `TSTORAGE_SIZE` and Address Clauses, appear to have no impact upon the compilation process.

The pragmas `Controlled`, `Inline`, `Interface`, `List (!)`, `Optimize`, `Pack`, `Page (!)`, `Shared` and `Suppress`, the length clauses, `TSIZE` and `TSMALL`, and Enumeration and Record Representation Clauses will impact upon the compilation process itself.

**PROVIDING EXPLICIT CONTROL OF SIZE OF MEMORY ACCESS,
I.E., BYTES, WORDS, LONG_WORDS.**

DATE: March 22, 1989

NAME: Ronald Sercely

ADDRESS: TASC
1250 Academy Park Loop, Suite 218
Colorado Springs, CO 80910

TELEPHONE: (719) 597-2222

ANSI/MIL-STD-1815A REFERENCE: 13.

PROBLEM:

The current reference manual provides no mention of control of memory access. On most architectures, such as the Motorola 680x0, DEC VAX, etc, memory may be accessed by byte, word or long-word. The difference can be crucial for memory mapped devices.

One example would be a 16 bit wide hardware FIFO that is memory mapped. Again, from first hand experience, compilers exist that generate twice the number of byte writes as desired, instead of the correct number of word writes. In my example, due to the hardware implementation, this resulted in the upper bytes having the correct (?) values, and the lower bytes being all "1"s. It was a very reasonable hardware implementation.

IMPORTANCE:

I believe this request is both administrative and important. It is administrative, because it is a technical correction that makes the standard more consistent with the design intent. Since all the effort to put record component representation clauses and address clauses into the language was done, why not complete the effort by allowing the user to specify the memory width, so that direct assignments can be accomplished? It is important in that it is not essential, (there are workarounds), and given the support for other representation clauses, it should have minimum negative impact to implementers.

CURRENT WORKAROUNDS:

The two expected workarounds do work, use of pragma interface (assembly) and use of package Machine_code.

POSSIBLE SOLUTIONS:

A new length clause should be provided, possibly in section 13.2, possibly section 13.5. For lack of a better name, I will call it memory_size. It partly fits into section 13.2 as a length clause, such as

for my_fifo'memory_size use 16;

-- or 2*byte if you prefer.

It does not fit here however, because all other length clauses are associated with a type, and this one would be associated with an object.

It fits into section 13.5 because 13.5 does refer to objects. In this section, it might appear as:

```
for my_fifo'memory_size use 16;
```

but this now conflicts with section 13.5(2), because my_fifo'memory_size is not a simple name.

Potentially a section 13.5.2 might be added, that would allow for a construct such as:

```
for my_fifo use width 16;
```

Adding this section would not affect any current numbering of reference manual entries. The use of "width" in the above is excellent for readability, but slightly conflicts with the language syntax, where the string in this field should probably be a reserved word. In this case, the construct could be:

```
memory_size_clause ::= for simple_name use mod simple_expression
```

resulting in:

```
for my_fifo use mod 16;
```

This is syntactically different enough from the other record representation clauses that confusion should not result.

Clearly usage of this feature would be implementation dependent, specifically, it must be restricted to the memory access capabilities of the underlying hardware.

ADD ATTRIBUTE TO ACCESS INTERNAL CODE OF ENUMERATION LITERAL**DATE:** February 15, 1989**NAME:** Arny B. Engelson**ADDRESS:** AT&T Bell Laboratories
Whippany Road (14B-257)
Whippany, NJ 07981**TELEPHONE:** (201) 386-4816
E-mail: att!wayback!arny**ANSI/MIL-STD-1815A REFERENCE:** 13.3**PROBLEM:**

There is no convenient language feature for determining what internal codes are being used to store enumeration literals. This would be a useful feature, and, given that a programmer may explicitly specify the internal codes to be used for enumeration literals by using an enumeration representation clause, it seems inconsistent to not allow convenient access to those specifications.

This presents a problem when values of the enumeration type must be sent between programs executing on different processors in a distributed application. We have such an application, where packetized data and control messages are sent between target processors and also between a host debugging system and the target processors. Common library packages are used to define enumeration values to be included as part of the messages. Enumeration representation clauses are used to map the internal codes to the bitwise specification of the message format. These messages are also created interactively at the host debugging station, as well as downloaded from a previously created file of messages.

The software that creates or interprets these messages needs a convenient (read efficient) mechanism to convert between an enumeration value and the (for example) 16 bit integer used as a field in a message.

IMPORTANCE:

An individual's scale of 1..10 tends to be very subjective. Therefore, I will only say the problem is not of major importance. But, in the determination of whether this feature should be added, I believe this is balanced by the simplicity of the proposed solution.

CURRENT WORKAROUNDS:

We currently solve the problem by instantiating `Unchecked_Conversion` for each enumeration type necessary, and converting the value to an appropriately sized integer. Besides the overhead in doing the instantiation, this is a messy solution. Other solutions (overlays, hard coded values) are equally unsatisfactory (erroneous or inefficient).

POSSIBLE SOLUTIONS:

A relatively simple, upwardly compatible solution, would be to add a new language attribute. Possibly as follows:

P'INTERNAL_CODE For a prefix that denotes an enumeration type or subtype:

This attribute is a function with a single parameter. The actual parameter X must be a value of the base type of P. The result type is the type universal integer. The result is the internal code used to represent X. (See 13.3.)

DEFAULT REPRESENTATION FOR ENUMERATION TYPES

DATE: October 28, 1988

NAME: S. Tucker Taft

ADDRESS: Intermetrics, Inc.
733 Concord Avenue
Cambridge, MA 02138

TELEPHONE: (617) 661-1840
E-mail: stt@inmet.inmet.com
E-mail: uunet!inmet!stt

ANSI/MIL-STD-1815A REFERENCE: 13.3(1)

PROBLEM:

The LRM should specify the default rep for enumeration types in the absence of a representation clause. Otherwise, every enumeration type which is to have the "conventional" rep (where the value = the 'POS) will need a rep clause if it is ever to be written out in binary form to the external environment. Similarly, the default rep for integer types should be specified as being unbiased (where the value = the 'POS).

IMPORTANCE:

Conservative programmers will continue to waste their energy, and very likely insert bugs due to typos, by spelling out enumeration representations which conform to the default representation.

Furthermore, there is some danger that an overeager compiler-writer will start representing enumeration types differently (e.g. using a signed representation), thereby destroying the portability of code which up until now was perfectly portable despite not going to the trouble of specifying default representations explicitly.

CURRENT WORKAROUNDS:

I have noticed in the code of certain very conservative Ada programmers that every enumeration type being passed via pragma Interface or written to a file, has a representation clause, even though the desired representation is exactly that implied by the definition of "POS" for the enumeration (i.e. starting at 0, going up sequentially).

This seems like a vast waste of effort since every Ada compiler under the sun uses the 'POS value as the default implementation. It seems perfectly reasonable to specify this as a requirement in the LRM.

Furthermore, the same argument could apply to integer types, but here there is no way for the user to control the representation, so it is even more important that there be some promise in the LRM that the default representation be unbiased/unshifted in the absence of a length clause.

In the presence of a length clause where only a biased representation would fit, then there should be some promise that the low bound of the type will be represented as a 0.

POSSIBLE SOLUTIONS:

Specify the default representation for enumeration and integer types explicitly, at least for values which pass out of the Ada "domain" via pragma Interface or binary input-output.

BIT/STORAGE UNIT ADDRESSING CONVENTION**DATE:** June 27, 1989**NAME:** Neil Salant**ADDRESS:** ITT Avionics
Dept. 73813
390 Washington Ave.
Nutley, NJ 07110**TELEPHONE:** (201) 284-3896**ANSI/MIL-STD-1815A REFERENCE:** 13.4**PROBLEM:**

Ada does not define a standard bit and storage unit addressing convention. The LRM specifies that "the ordering of bits in a storage unit is machine dependent and may extend to adjacent storage units". This results in portability and communications problems between big endian and little endian bit addressed machines. For example, on a MIL-STD-1750A processor, the sign bit is bit 0 and the least significant bit is number 15. On a VAX, the sign bit is number 15 (for 16 bit integers) and the least significant bit is number 0. This if we need to access the high order nibble of a word, on a 1750 machine we specify bits 0 through 3 while on a VAX we specify bits 12 through 15. Porting software from one machine to the other would require changing all bit addresses to the appropriate values. Similarly, Ada does not define a word addressing convention, which becomes problematic when components extend into adjacent storage units. In some machines, the most significant part is the higher addressed storage unit, while in other machines the most significant part is the lower addressed storage unit.

IMPORTANCE: IMPORTANT

The lack of an Ada bit/storage unit addressing convention imposes portability constraints for applications which utilize record representation clauses or "bit arrays" to interface with hardware or to other computer systems that have different processor/Ada compiler addressing conventions. It is also required for projects that follow a "software first" approach (i.e. initially develop code on a host processor and then port code to the target system) when the host and target conventions are different.

CURRENT WORKAROUNDS:

1. An application may decide to conform to either big or little endian bit addressing convention, and calculate all possible single bit addresses, as well as bit field start and stop addresses during system initialization. This method will work only if the application is able to determine what processor it is executing on by examination of package SYSTEM. Then, knowing that the processor either big endian or little endian bit addressed, it calculates $n**2$ constant values for a bit range n . These values give the proper start and end for all possible multiple bit fields and single bit fields. If all references to bit fields within record representation clauses and bit arrays use these $n**2$ constant values for bit addressing, then Ada code will correctly port between big and little endian bit addressed processors. While somewhat awkward, this method is usable for 16 bit processors, requiring 256 constant values. For 32 bit processors, 1024 constants must be declared and initially calculated.

2. A tool can be implemented which parses Ada source code, looking for bit address instances. When a bit address is located, the value is changed to the correct bit address for a given target processor.

POSSIBLE SOLUTIONS:

Define a bit addressing convention for the Ada language and let the compiler resolve the compiler resolve bit/storage unit addresses for processors which do not conform to the Ada convention.

PROVIDE EXPLICIT CONTROL OF MEMORY USAGE**DATE:** June 7, 1989**NAME:** Offer Pazy (ACM Special Interest Group on Ada, Ada Runtime Environment Working Group)**ADDRESS:** Software Leverage Inc.
485 Massachusetts Ave.
Arlington, MA 02174**TELEPHONE:** (617) 648-1414
E-mail: offer@sli.com
E-mail: pazyo@ajpo.sei.cmu.edu**ANSI/MIL-STD-1815A REFERENCE:** 13.5**PROBLEM:**

The language makes no distinction between physical addresses and logical addresses, nor does it provide for systems with mixed memories (e.g. RAM and ROM) or fragmented non-contiguous memories. The current Ada memory model works well for general purpose computing systems but lacks the needed expressiveness for embedded real-time computing systems, which must assign Ada objects to memory in terms of location (both absolute physical location and relative location within a logical address space), range, and type of memory.

Ada provides facilities for specifying storage locations for certain code and data objects (ARM 13.5), and for specifying an object's size (ARM 13.2), but these facilities are not sufficient to meet all of the requirements for embedded, real-time, systems.

SPECIFIC REQUIREMENTS/SOLUTION CRITERIA:

Requirements for control of memory allocation in embedded, real-time, systems include:

- 1) The capability to explicitly direct the assignment of code and/or data objects to specified real memory regions. The language should provide a mechanism to explicitly assign data objects and the code of packages, library units, and task bodies to specified memory regions.
- 2) The capability to specify the storage characteristics of any collection, package, library unit or task, e.g. read/write or read only.
- 3) Specification of the storage for the main program and tasks' stacks should similarly be under program control.

IMPORTANCE:

As long as there are no clear and standard methods to handle the fragmented and heterogeneous memory issues for Ada programs, contractors will perceive this inadequacy as a justification to obtain programming language waivers for mission critical computer systems.

CURRENT WORKAROUNDS:

The following are examples of memory requirements found in current embedded real-time systems:

- 1) Avionic systems require hardware-vendor supplied Built-In-Test (BIT) software and other ROM based module-specific functions that must be executed by the operational flight software. Therefore, a "reserve memory" capability is necessary to direct the Ada compilation system not to use a certain range of addresses.
- 2) Many embedded systems use on-line databases that are kept in UVROM or EEPROM so they are not subject to loss by power transients. The program needs to have control over the allocation of such memory regions.
- 3) Tightly-coupled multi-processor systems that have global memory, often place shared segments at memory addresses that are not contiguous with per processor system memory.

Since hardware requirements for embedded, real-time, systems often mandate that code and/or data reside in RAM or ROM, and must be located at specific physical locations within the system, there is a need for better specification mechanisms in the language than currently exist. Historically, linkers have been used to provide the physical placement of code and data objects, but linkers are not always capable, nor do they necessarily get all the information required from the compiler. Besides, linkers are beyond the scope of the language and this affects the reusability and maintainability of the Ada programs.

DIFFICULTIES TO BE CONSIDERED:

There is a real requirement for explicit control of memory allocation, however, the solution may lie outside the scope of the Ada language.

The solution may be machine-dependent and/or implementation-dependent. An acceptable solution may call for an ancillary standard per machine architecture.

ADDRESS CLAUSES AND INTERRUPT ENTRIES

DATE: June 7, 1989

NAME: Tom Quiggle (ACM Special Interest Group on Ada, Ada Runtime Environment Working Group)

ADDRESS: TeleSoft
5959 Cornerstone Court West
San Diego, CA 92121

TELEPHONE: (610) 457-2700
E-mail: quigglet@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 13.5(6)

PROBLEM:

Many applications are required to handle interrupts from multiple identical devices. An interrupt handler in Ada is written as a task. The most natural way to write interrupt handlers for more than one identical device is to declare multiple objects of the same task type. This cannot be done as the address clause of an entry is associated with the type of the task rather than each object of the task type. Hence the address clause is evaluated once, when the type comes into scope.

SPECIFIC REQUIREMENT/SOLUTION CRITERIA:

It should be possible to associate multiple objects of the same task type with multiple interrupting devices of the same kind.

IMPORTANCE:

Applications interfacing to multiple instances of the same interrupting device will require the artificial introduction of multiple, distinct, task types. The introduction of distinct types precludes passing the resulting task objects interchangeably as parameters to subprogram and entry calls, and increases the likelihood that the code for the task types will be replicated.

CURRENT WORKAROUNDS:

The workaround is to create distinct task types for each interrupting device. This can be simplified with the use of generics by placing the task type declaration in a generic package with a generic formal object of type System.Address used as the expression in the address clause for the interrupt entry. E.G.

```
generic
  INT_SOURCE : SYSTEM.ADDRESS;
package DECLARES_HANDLER is

  task type HANDLER is
    entry INT_ENT;
    for INT_ENT use at INT_SOURCE;
  end HANDLER;
```



```
HANDLER_TASK : HANDLER;

end DECLARES_HANDLER;

...

package HANDLER_40 is new DECLARES_HANDLER( 16#40# );
package HANDLER_44 is new DECLARES_HANDLER( 16#44# );
package HANDLER_48 is new DECLARES_HANDLER( 16#48# );
-- Assuming that System.Address is an integer type.

...
```

POSSIBLE SOLUTIONS:

One solution is to provide a default initialization mechanism for task types (similar to that for record types) which is evaluated once for each object declared of the task type, and allow the association of task entries with interrupt sources via such initialization.

Alternately, the language could provide a construct which appears in the body of a task unit to associate a task entry with an interrupt source. One possibility would be to permit an address clause for a task entry to appear in a task body.

DIFFICULTIES TO BE CONSIDERED:

Allowing address clauses in a task body would require redefinition of forcing occurrences, and may have significant impact on separate compilation issues.

MODELS FOR INTERRUPT HANDLING

DATE: June 7, 1989

NAME: Mike Kamrad (ACM Special Interest Group on Ada, Ada Runtime Environment Working Group)

ADDRESS: Unisys Computer Systems Division
MS Court
PO Box 64525
St. Paul MN 55164-0525

TELEPHONE: (612) 456-7315
E-mail: mkamrad@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 13.5.1

PROBLEM:

The interrupt handling mechanism defined by the Ada Reference Manual (ARM) should model real interrupt handlers. Current implementations provide inconsistent and often inadequate support (and documentation) for the mechanisms used to handle interrupts in Ada; causing user confusion over how interrupt handling should be correctly programmed in Ada applications. As an example of an approach to eliminate these problems, a model is proposed that describes the possible connections between a hardware interrupt and a corresponding interrupt handler.

SPECIFIC REQUIREMENT/SOLUTION CRITERIA:

A programmer needs to know how to write an interrupt handler in Ada such that the execution of that interrupt handler can be predicted.

IMPORTANCE:

Continued confusion over how to program interrupt handling in Ada will cause either the avoidance of Ada interrupt entries, incorrect usage of tasks as interrupts handlers, or the proliferation of nonstandard approaches for handling interrupts.

CURRENT WORKAROUNDS:

Due to inconsistent support for interrupt handling, ambiguities and misunderstandings concerning programming of Ada interrupts have proliferated. In particular, the Ada concept of an interrupt task can be interpreted as either a mapping to the traditional interrupt procedure implementation, or as the invocation of a handler task in response to an interrupt. Problems have arisen due to the varying interpretations of these handler constructs by vendors, and by a misunderstanding on the part of users regarding what the vendors have implemented.

As an example of an approach to eliminate these problems, a model for describing an implementation of interrupt handling is provided by the referenced paper. This model describes the handling of interrupts in terms of a two level process, with either direct or indirect coupling between the levels. The first level interfaces directly with the hardware; i.e., typically saving processor state and performing runtime system processing. The second level deals with application-level interrupt processing. Direct coupling between the

levels is implemented as a procedure call, while indirect coupling is implemented through a task scheduling mechanism. In this canonical model, there are four alternatives for handling interrupts:

- 1) Single-Level Interrupt Processing: Direct connection between a hardware interrupt and a handler task accept body. The accept body will be the interrupt procedure, which must perform (or call predefined routines for) state saving [and restoring] as well as the handling of the specific interrupt.
- 2) Two-Level Interrupt Processing with Direct Coupling: Direct connection between a hardware interrupt and an runtime system routine which must provide state saving [and restoring], and which will call the accept body. The accept body is treated as a procedure.
- 3) Two-Level Interrupt Processing with Indirect Coupling: Direct connection between a hardware interrupt and an runtime system routine which provides state saving [and restoring], and which schedules a handler task containing the accept body. In this case the accept body is treated as a normal Ada task.
- 4) Two-Level Interrupt Processing with Direct and Indirect Coupling: Direct connection between a hardware interrupt and an runtime system routine which provides state saving [and restoring], and which will call the accept body (as in case 2, above). In addition, however, "simple" calls to task entries from within the accept body will cause scheduling of the associated tasks, and, upon return from the interrupt handler, dispatching of these tasks.

In addition to the general problem of vendor-to-user communication, the following, more specific, problems have also been identified:

- a) Interpretation of priorities for interrupt handler tasks.
- b) Lack of clarification of how conditional entry calls to interrupt handler tasks are to be treated.
- c) Lack of suitable provisions for tailoring the runtime system interrupt handling "sequence" to accommodate special state saving and restoring requirements (e.g. to save and restore the state of coprocessors).

POSSIBLE SOLUTIONS:

Solutions can include any or all of the following overlapping alternatives:

- 1) It is desired to extend the ARM to include a detailed description of a two-level interrupt handling model. Compiler implementors could then be required to describe their implementation of interrupt handling in terms of this model.
- 2) A detailed description of a two-level interrupt handling model should be specified as an ancillary standard. This standard may be expanded to include adaptations for each instruction set architecture.
- 3) Additional predefined pragmas or representation clauses (or other language features) could be added to the language to allow the application builder to clearly specify the required interrupt handler implementation(s).

REFERENCES/SUPPORTING MATERIAL:

- [1] C. Malec, "Interrupts" Boeing Advanced Systems, Seattle, WA Release 1.0, 24 October 1988.

TASKING PRIORITY INVERSION BECAUSE OF HARDWARE INTERRUPT**DATE:** May 10, 1989**NAME:** Robert Page**ADDRESS:** Commander (Code 3922)
Naval Weapons Center
China Lake, CA 93555-6001**TELEPHONE:** (619) 939-1288**ANSI/MIL-STD-1815A REFERENCE:** 13.5.1(2)**PROBLEM:**

Modeling an interrupt as an entry call issued by a hardware task whose priority is higher than the priority of the main program or tasks, is too restrictive. Why should a low priority, soft deadline device, for example a keyboard input, interrupt a high priority, hard deadline task, such as one controlling the stability of an aircraft?

There should be an integrated and consistent treatment of software and hardware priorities in Ada. If there are i levels of interrupt priority and j levels of software (tasking) priority, the i levels of interrupt priority are currently considered to be higher than the software priorities. In the future, we should allow the user to provide a mapping such as the following (for $i=3$ and $j=10$):

Overall priority	Hardware priority	Software priority
1		1
2		2
3		1
4		3
5		4
6		2
7		5
8		6
9		3
10		7
11		8
12		9
13		10

If the number of levels of interrupt priority equals the number of levels of software priorities, then the mapping could be one-to-one. The user defined mapping could be defined to implement the current definition in 13.5.1(2) by mapping the interrupt priorities to be higher than the software priorities.

IMPORTANCE: ESSENTIAL

If this request is not satisfied, I could not support acceptance of the revised standard. The dangers of priority inversion in an embedded real-time system have already been documented. ("Real-Time Scheduling Theory in Ada", Sha and Goodenough, SEI document CMU/SEI-88-TR-33, pp. 2, 17-20).

CURRENT WORKAROUNDS:

One workaround ("Real-Time Scheduling Theory in Ada", Sha and Goodenough, SEI document CMU/SEI-88-TR-33, pp. 19) is to remove as much functionality as possible from the interrupt handler (to keep the blocking from the interrupt as short as possible) and place that functionality in yet another task of the appropriate priority.

POSSIBLE SOLUTIONS:

SETTING/ADJUSTING CALENDAR.CLOCK

DATE: June 7, 1989

NAME: Ted Baker (ACM Special Interest Group on Ada, Ada Runtime Environment Working Group)

ADDRESS: Department of Computer Science
Florida State University
Tallahassee, FL 32306-4019

TELEPHONE: (904) 644-5452
E-mail: (ARPAnet) tbaker@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 13.7

PROBLEM:

The language should define a standard way for application programs to set and adjust CALENDAR.CLOCK, in order to synchronize it with other time references. Ada currently provides no such mechanism.

SPECIFIC REQUIREMENT/SOLUTION CRITERIA:

A standard way should be provided for an application to set the clock.

Both relative and absolute time adjustments must be supported; that is, it must be possible to both set the clock to an absolute time, and to adjust it forward or backward by some duration.

It is desirable that relative time adjustments be "smoothed". For example, if one recognizes that the local clock is 400 ticks ahead of "real-time", one may wish to have the clock implementation skip one out of every 10 ticks, until it catches up.

IMPORTANCE:

If no standard mechanism is provided to adjust the clock, application builders will force one, by modification of the Ada RTE code. This requires an intimate knowledge of the particular Ada RTE implementation; information that is often hard to obtain. The repeated effort of doing these custom modifications for each application is wasteful, and the results are generally not portable or re-usable across applications, compilers, or targets.

CURRENT WORKAROUNDS:

The ability to modify the clock value is a very common requirement in real-time embedded systems. There is no good reason why it cannot be done in a standard way.

A partial list of reasons for clock adjustments are:

- 1) to correct for clock drift among multiple processors (as often as once every few seconds);
- 2) to align with a new time zone (for platforms that move and do not operate on Universal Time);

- 3) to adjust for "Daylight savings";
- 4) to add "leap" seconds.

The kind of clock adjustments, the time references on which they are based, and the method of resolving conflicts between multiple time references, are details that vary among different embedded applications.

Smoothing is especially important for backward adjustments of the clock, since sudden decrements in the time value can be fatal for algorithms such as periodic trajectory planners, where extrapolation is used. If an adjustment to the clock is made instantaneously, the error introduced is likely to exceed the error budget. Speeding up the clock is a way to gradually bring the system back into synchronization without being too disruptive.

POSSIBLE SOLUTIONS:

Adjustment procedures could be added to the package CALENDAR. For example:

```
procedure SET_CLOCK(NEW_TIME: TIME);  
  
procedure SET_CLOCK(TOTAL_ADJUSTMENT: DURATION;  
                    ADJUSTMENT: DURATION := 0.0;  
                    ADJUSTMENT_PERIOD: DURATION := 0.0);
```

For the second procedure, if either ADJUSTMENT or ADJUSTMENT_PERIOD are zero, the adjustment occurs immediately. Otherwise, ADJUSTMENT is added to the CLOCK time immediately and every ADJUSTMENT_PERIOD thereafter until the TOTAL_ADJUSTMENT has been made. If TOTAL_ADJUSTMENT is not a multiple of ADJUSTMENT, the last adjustment will be smaller than ADJUSTMENT. If another SET_CLOCK (relative or absolute) is issued prior to completion of an adjustment, the new SET_CLOCK will take over where the old SET_CLOCK left off. Specifically, the adjustment variables will be set strictly by the new SET_CLOCK.

An alternate method would be for the RTE to apply a standard smoothing algorithm to relative adjustments.

DIFFICULTIES TO BE CONSIDERED:

Adjusting the system clock is a privileged operation on most multiuser systems. If a program without permission tries to adjust the clock, the operation should be permitted to raise an exception.

USE OF ASSEMBLY LANGUAGE

DATE: February 21, 1989

NAME: SY Wong

ADDRESS: 5200 Topeka Drive
Tarzana, CA 91356

TELEPHONE: (818) 345-6274
E-mail: hermix!sywong@rand-unix.arpa

ANSI/MIL-STD-1815a REFERENCE: 13.8, 13.9

PROBLEM:

These sections hinder Portability. Chapter 13.8 "Machine language" is in reality binary code insertion and is easily misinterpreted to be assembly language. Assembly language properly belongs to "other language" in 13.9. Rationale 15.7 may not be correct in stating that "Furthermore, its (machine language) use is heavier than direct use of an assembler: the facility offered should be sufficient for cases where there is an actual need, but its style will inhibit overuse."

This statement does not reflect real world situations. The net result is the stifling of the judicious use of assembly language in time critical situations to supplement Ada deficiencies.

13.9 "Other languages" appears to be intended for implementer supplied libraries. There is no package body required. The requirement to state pragma interface in the private section is probably intended for the convenience of the compiler writer with no portability and implementation hiding in mind. The specification should not be implementation specific.

Neither section adequately considered application programmer generated assembly, or other language in package bodies, with adequate source documentation.

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS:

One validated compiler vendor offers assembly language facilities interfaced to clean Ada specifications without pragma interface, for his compiler. This is legal, and desirable, but is not a portable solution.

POSSIBLE SOLUTIONS:

1. In 13.9, disallow pragma interface in the specification. The body should [allow] the inclusion of "other language" source codes (see work around), with reference to language syntax and semantics specifications. Assembly language should be clearly stated as a "foreign language", thereby bypassing the awkward 13.8.
2. (Optional) remove section 13.8.
3. Rationale 15.7 should be reworded to encourage the judicious use of assembly language packages (not code insertion) in critical situations.

GUARANTEE MEMORY RECLAMATION

DATE: June 7, 1989

NAME: Offer Pazy (ACM Special Interest Group on Ada, Ada Runtime Environment Working Group)

ADDRESS: Software Leverage Inc.
485 Massachusetts Ave.
Arlington, MA 02174

TELEPHONE: (617) 648-1414
E-mail: pazyo@ajpo.sei.cmu.edu
E-mail: offer@slf.com

ANSI/MIL-STD-1815A REFERENCE: 13.10.1

PROBLEM:

Various applications (such as real-time, long running, or embedded systems) require that unused memory is always reclaimed. The Ada standard does not provide for uniform portable means to guarantee that these requirements are met.

SPECIFIC REQUIREMENTS/SOLUTION CRITERIA:

The language shall require that a conforming compiler/runtime shall never "lose" storage. This shall include not only storage allocated by allocators, but also storage for task objects that have terminated and are no longer accessible, as well as storage allocated by the compiler/runtime for their own use. Moreover, upon completion of an Ada program, all storage consumed by the program shall be returned to any underlying operating system and/or executive.

The Language shall state that, in the absence of an explicit application request to the contrary, storage occupied by an object which was created by an allocator shall be reclaimed, e.g. made eligible for garbage collection. In particular, this is important in certain error conditions when memory is allocated but cannot be referenced.

IMPORTANCE:

Without a guaranteed deterministic memory reclamation strategy, in particular for memory that is being used implicitly by the implementation, long-running applications will find it difficult to use features of the Ada language such as tasking and allocators.

CURRENT WORKAROUNDS:

Certain Ada applications must run for arbitrarily long periods of time. For such applications, storage management is obviously critical. Given a long enough period of time, the loss or mishandling of even small amounts of storage could lead to an application failure.

A special problem exists with tasks which depend on library packages, because their master is not exited prior to program termination. In the current language, it is not possible to reclaim all the storage associated with the task. (The note in 13.10.1(8) says that `UNCHECKED_DEALLOCATION` on a task

object has no effect). Even when a task created via an allocator has terminated, it is still accessible via the associated access value. The result is that highly dynamic task allocation will eventually result in storage error.

In addition, certain error conditions can leave dangling unused blocks of memory which are invisible to the programmer and thus cannot be freed explicitly. For example, consider the following:

```
package PKG1 is
  -- A library package

  type REC1 is
    record
      I1 : INTEGER;
    end REC1;

  type P1 is access REC1;

  type REC2 is
    record
      J2 : P1 := new REC1;
    end record;

  type P2 is access REC2;

end PKG1;

with PKG1; use PKG1;
procedure PROC is
  OBJ : P2;
begin
  OBJ := new REC2;
exception
  when STORAGE_ERROR =>
    null; -- ??
end PROC;
```

While processing the "new" statement of REC2, there may be a STORAGE_ERROR raised while trying to allocate memory for REC1. In that case the memory allocated for REC2 can be left hanging. The programmer cannot deallocate it (the assignment to Obj has not completed), and since the access types are declared in a library package, even an "aggressive" implementation cannot deallocate the inaccessible memory.

The problem may be further complicated if REC1 contains task objects. In that case these tasks and the memory associated with them can be left hanging with no way to either name them, abort them or allow them to "normally" terminate.

POSSIBLE SOLUTIONS:

- 1) The generic function UNCHECKED_DEALLOCATION should be required to either fully reclaim memory or to be clearly rejected (i.e the phrase allowing this function to be ignored should be removed). This function should also be required to work on tasks (at least terminated).

- 2) Disallow implementations to leave chunks of memory blocks dangling as a result of certain error conditions.

DIFFICULTIES TO BE CONSIDERED:

This requirement may not be fully implementable given the current Ada language definition. If some of the rules concerning record initialization with allocators are changed, this requirement would be easier to implement.

REFERENCES/SUPPORTING MATERIAL:

- [1] A. J. Wellings, et. al. "A Problem with Ada and Resource Allocation", Ada Letters, Vol. 3 No. 4, January/February 1984.

LIMITATIONS OF UNCHECKED CONVERSION

DATE: May 17, 1989

NAME: T. P. Baker

ADDRESS: Department of Computer Science
207A Love Building
Florida State University
Tallahassee, FL 32306-4019

TELEPHONE: (904) 644-5452
E-mail: tbaker@ajpo.sei.cmu.edu
E-mail: baker@nu.cs.fsu.edu

ANSI/MIL-STD-1815A REFERENCE: 13.10.2

PROBLEM:

Ada does not adequately support the type conversions needed to interface to operating systems, mass storage devices, and other non-Ada sources of input-output. This problem comes up in a number of contexts, including interfacing to operating systems (e.g. POSIX), and writing table-driven programs.

Because unchecked conversions are formally function calls (not type conversions), unchecked conversions cannot be used on OUT and IN OUT parameters. This forces wasteful copying of data through intermediate variables.

For large data objects, copying actually becomes impossible, due to storage and addressing limitations. Some form of data blocking is necessary.

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS:

The safest workaround is to use an intermediate variable, as sketched below:

```
package POSIX_IO is
  type BYTE is private;
  function READ(B: out BYTE);
end POSIX_IO;

with POSIX_IO;
with UNCHECKED_CONVERSION;
procedure P is
  C: CHARACTER;
  B: POSIX_IO.BYTE; -- intermediate variable
  function FORCE is new UNCHECKED_CONVERSION(POSIX_BYTE,CHARACTER);
begin
  READ(B); C:=FORCE(B);
end P;
```

This solution is inefficient. Worse, it breaks down entirely if the object being copied is very large.

I have had this problem when I write table-driven parsers and code-generators. The tables are around 100K bytes of binary data, which are organized into several substructures, including arrays of records. I want to read them in as quickly as possible. I tried making the tables into a single record, declaring a sequential file of this record type, and using SEQUENTIAL_IO. Naturally enough, Ada implementations try to allocate a buffer big enough to hold at least one record. This is too big, so the program failed.

I ended up doing something like this, using unchecked conversion on access types and addresses:

```

type PARSING_TABLE is
  record
    ENDMARKER    : ....;
    LRHS         : array (....) of ....;
    ACTION       : array (....) of ....;
    ....
  end record;

TABLE: PARSING_TABLE;

type BLOCK is array (0..1023) of INTEGER;
package BLOCK_IO is new SEQUENTIAL_IO(BLOCK);

type TABLE_ALIAS is array(1..TABLE'size/BLOCK'size) of BLOCK;
type ALIAS_PTR is access TABLE_ALIAS;
function FORCE is new UNCHECKED_CONVERSION(SYSTEM.ADDRESS,ALIAS_PTR);
ALIAS: ALIAS_PTR:= FORCE(TABLE'address);

procedure READ_PARSING_TABLE is
  TABLE_FILE: BLOCK_IO.FILE_TYPE;
begin
  if TABLES_INSTALLED then return; end if;
  begin
    BLOCK_IO.OPEN(TABLE_FILE,IN_FILE,"tables");
    for I in TABLE_ALIAS'range loop
      BLOCK_IO.READ(TABLE_FILE,ALIAS.all(I));
    end loop;
  exception when others =>
    QUIT("table file does not exist");
  end;
  TABLES_INSTALLED:= TRUE;
end READ_PARSING_TABLE;
```

This presumes access values are implemented as addresses -- which is frequently wrong! The code is therefore not portable. Aside: This example brings up another sore point -- There ought to be a better way than raising an exception to find out whether a file exists and can be opened in a given mode!

POSSIBLE SOLUTIONS:

A partial solution (for small structures) would be to add a new syntactic form for unchecked conversions, that can be applied to OUT and IN OUT parameters. This form should be of the category "name".

The root of the problem seems to be the definition of unchecked conversion as a generic function. Note that it is not possible to allow general function calls as OUT or IN OUT parameters, but type conversions are different from arbitrary functions, in that they are invertible. In fact, unchecked conversions are especially so, since they should not involve any processing.

This still does not solve the problem of how to block large structures into smaller ones. For this, some explicit form of overlaying seems necessary.

This might be achieved by relaxing the restriction on the use of address clauses to achieve overlaying, and requiring they be supported for general (not just static) address expressions.

Then, for example, the following solution might work:

```
procedure READ_PARSING_TABLE is
  type TABLE_ALIAS is array(1..TABLE'size/BLOCK'size) of BLOCK;
  ALIAS: TABLE_ALIAS;
  for ALIAS use at TABLE'address;
  TABLE_FILE: BLOCK_IO.FILE_TYPE;
begin
  .... BLOCK_IO.READ(TABLE_FILE,ALIAS(I)); ....
end READ_PARSING_TABLE;
```

Since I don't like using global variables, I would prefer this be required to work also for OUT and IN OUT parameters, e.g. procedure READ_PARSING_TABLE(TABLE: out PARSING_TABLE) is

```
  ALIAS: TABLE_ALIAS;
  for ALIAS use at TABLE'address;
  ...
```

This feature would probably eliminate the need for unchecked conversions.

A disadvantage is that there is no longer clear visibility that something unsafe is going on (as via "with UNCHECKED_CONVERSION"). This weakness could be corrected by tying this feature to some library unit, just as using 'ADDRESS currently requires package SYSTEM. [ada9x]

For additional references to Section 13. of ANSI/MIL-STD-1815A, see the following sections, revision request numbers, and revision request titles in this document.

REVISION REQUEST

<u>NUMBER</u>	<u>TITLE</u>	<u>SECTION</u>
0024	SEPARATION OF EXPONENT AND MANTISSA	4
0109	DISTRIBUTED SYSTEMS	9
0037	CONTROL OF CLOCK SPEED AND TASK DISPATCH RATE	9
0107	CONFIGURING CALENDAR.CLOCK IMPLEMENTATION	9

ADA 92 REVISION REQUESTS
THAT REFERENCE
ANSI MIL-STD-1815A

SECTION 14. INPUT-OUTPUT

ADD PREDEFINED KEYBOARD I/O PACKAGE**DATE:** February 20, 1989**NAME:** Chris Clarke**ADDRESS:** Berkshire, Fox & Associates
Post Office Box 90673
Pasadena, CA 91109-0673**TELEPHONE:****ANSI/MIL-STD-1815A REFERENCE:** 14.**PROBLEM:**

The current version of Ada does not specify a predefined keyboard input-output package. As a result, each time an Ada program is ported to a new machine, the keyboard input-output package must be completely rewritten. In many cases, the keyboard input-output package is the only part of the program which must be rewritten. Designing, coding, and testing a new keyboard input-output package can be a tedious and time consuming process - which can significantly increase the cost to port Ada programs from one machine to another.

IMPORTANCE: ADMINISTRATIVE

If this request is not satisfied, then users of Ada will continue to incur extra costs to port programs from one machine to another.

CURRENT WORKAROUNDS:

A custom keyboard input-output package is either written by the programmer or supplied by the compiler manufacturer.

POSSIBLE SOLUTIONS:

Add a new predefined package to Ada called `KEYBOARD_INPUT`. This new package would have one procedure called `GET_KEY_VALUE`. The procedure `GET_KEY_VALUE` could either be generic for appropriate types, or, alternatively, return a one character element of type `CHARACTER` (which would be a new, expanded type `CHARACTER` derived from the Extended ASCII character set).

INTERACTIVE TERMINAL INPUT-OUTPUT

DATE: March 1. 1989

NAME: Mike Curtis (and members of the Ada-Europe Environments Working Group)

ADDRESS: ICL
Eskdale Road
Winnersh
Wokingham
Berkshire RG11 5TT
UK

TELEPHONE: +44 734 693131

ANSI/MIL-STD-1815A REFERENCE: 14.

PROBLEM:

The facilities for Input-Output provided by the `TEXT_IO` package are inadequate for handling modern terminals and programming the sort of user-interfaces expected in modern software packages.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

One possibility is that the low-level functions of the terminal, for example escape codes for screen control and values returned by function keys, can be defined, and then `TEXT_IO` can be used. This can give some limited control but it becomes extremely complex if terminal independence is required and it gives no help with use of the graphics characters for line drawing.

The other possibility is to use a different language.

POSSIBLE SOLUTIONS:

A new package `"TERMINAL_IO"` can be defined, with the same status as `TEXT_IO` in that it can be omitted if it is not relevant for a particular compiler/machine combination. It should include basic screen operations, such as positioning the cursor and clearing and scrolling all or part of the screen, recognition of function keys and possibly higher level operations such as the creation and manipulation of text windows. Provision should be made for the use of the full extended character set.

DO NOT ADD VARIABLE STRING TYPE

DATE: March 9, 1989

NAME: SY Wong

ADDRESS: 5200 Topeka Drive
Tarzana, CA 91356

TELEPHONE: (818) 345-6274
E-mail: hermix!sywong@rand-unix.arpa

ANSI/MIL-STD-1815a REFERENCE: 14.

PROBLEM:

Inadequately considered demands for variable string types.

A variable string is a cure that is worse than the disease because now we have to deal with two string types, affecting portability and tool interoperability.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

Ada string with slice is somewhat cumbersome but adequate. It only require some care in using unconstrained string parameters and return types in subprogram declarations.

POSSIBLE SOLUTIONS:

Do not add a variable string type in the language.

If 9X is stamped to add variable string type, the logical place to add is in text_io, with overloaded operations. Strings are often used in conjunction with text_io.

FILE AND RECORD LOCKING

DATE: June 22, 1989

NAME: Chris Clarke

ADDRESS: Berkshire, Fox & Associates
Post Office Box 90673
Pasadena, CA 91109-0673

TELEPHONE:

ANSI/MIL-STD-1815A REFERENCE: 14.2.3, 14.2.5

PROBLEM:

Ada does not currently offer a way to lock files or records. As a result, when record or file locking is required, programmers must insert custom assembly language code into their Ada programs. Designing, coding, and testing this custom code can be a tedious and time consuming process - which can significantly increase the cost of the whole programming effort.

IMPORTANCE: ADMINISTRATIVE

If this request is not satisfied, then, users of Ada will continue to incur extra costs to create programs which lock files or records.

CURRENT WORKAROUNDS:

Programmers must insert custom assembly language code into their Ada programs.

POSSIBLE SOLUTIONS:

In order to lock files, add a new predefined procedure to the Sequential_IO and Direct_IO packages called OPEN_AND_LOCK. The file would then be unlocked when it is closed. In order to lock and unlock records, add two new predefined procedures to the Direct_IO package called READ_AND_LOCK and RELEASE.

"GET" AND "PUT" AS FUNCTIONS

DATE: March 4, 1989

NAME: William Thomas Wolfe

ADDRESS: Home: 102 Edgewood Avenue #2
Clemson, SC 29631 USA

Office: Department of Computer Science
Clemson University
Clemson, SC 29634 USA

TELEPHONE: Home: (803) 654-7030
Office: (803) 656-2847
E-mail: wtwolfe@hubcap.clemson.edu

ANSI/MIL-STD-1815A REFERENCE: 14.3.6(7-9), 14.3.7(13-17), 14.3.8(17-21), 14.3.9(10-15)

PROBLEM:

The GET and PUT procedures defined in TEXT_IO return their results as out parameters. This forces one to define a fixed-size string, for example, to contain the result. However, one cannot necessarily be certain how long a string will be required to hold the result.

CONSEQUENCES:

The programmer must supply a container of some arbitrary size and hope that the size of the container will always be adequate.

CURRENT WORKAROUNDS:

Supply an extremely large container, space requirements be damned.

POSSIBLE SOLUTIONS:

Revise ANSI/MIL-STD-1815A 14.3.6 (7-9), 14.3.7 (13-17), 14.3.8 (17-21), and 14.3.9 (10-15) such that GET and PUT are functions rather than procedures. This way, one can receive a STRING of arbitrary length and send it directly into some variable-length string type's assignment procedure, thus removing the need to specify the size of an intermediate container of type STRING.

Example:

```
VARIABLE_LENGTH_STRINGS.ASSIGN (THE_RESULT, TEXT_IO.PUT (SOME_FLOAT)); or if we  
get a "real" assignment operator,  
THE_RESULT := TEXT_IO.PUT (SOME_FLOAT);
```

Another possible solution would be to define 'IMAGE for non-discrete types, such as universal_real and its subtypes (ANSI/MIL-STD-1815a reference: A (18)).

DEFINE "DEFAULT_XY" IN IO PACKAGES AS FUNCTIONS**DATE:** May 23, 1989**NAME:** Jurgen F H Winkler**ADDRESS:** Siemens AG ZFE F2 SOF3
Otto-Hahn-Ring 6
D-8000 Munchen 83
Fed Rep of Germany**TELEPHONE:** +49 89 636 2173**ANSI/MIL-STD-1815A REFERENCE:** 14.3.7, 14.3.8, 14.3.9**PROBLEM:**

Define the entities named "DEFAULT_xy" as functions

The variables DEFAULT_FORE etc. in the packages for integer IO, real IO, and enumeration IO can be altered by assignment statements. Other similar entities in the IO packages, e.g. default input, current input, or current line, are defined as parameterless functions and can only be altered by corresponding procedures as e.g. SET_COL.

IMPORTANCE: ADMINISTRATIVE**CURRENT WORKAROUNDS:****POSSIBLE SOLUTIONS:** Obvious

OUTPUT OF REAL NUMBERS WITH BASES**DATE:** May 23, 1989**NAME:** Jurgen F H Winkler**ADDRESS:** Siemens AG ZFE F2 SOF3
Otto-Hahn-Ring 6
D-8000 Munchen 83
Fed Rep of Germany**TELEPHONE:** +49 89 636 2173**ANSI/MIL-STD-1815A REFERENCE:** 14.3.8**PROBLEM:**

Allow output of real numbers with bases from 2 to 16

Ada allows the notation of real numbers with bases from 2 to 16 in the source program and in the input for the GET procedures, but does not allow the specification of a base in the PUT procedures.

IMPORTANCE: ADMINISTRATIVE**CURRENT WORKAROUNDS:**

Write new PUT procedures.

POSSIBLE SOLUTIONS:

subtype NUMBER_BASE is INTEGER range 2..16;

DEFAULT_BASE : NUMBER_BASE: = 10;

```
procedure PUT ( FILE      :in FILE_TYPE;
                ITEM      :in NUM;
                FORE      :in FIELD: = DEFAULT_FORE;
                AFT       :in FIELD: = DEFAULT_AFT;
                EXP       :in FIELD: = DEFAULT_EXP;
                BASE      :in NUMBER_BASE: = DEFAULT_BASE);
```

**SECTION 15. REVISION REQUESTS
THAT REFERENCE
ANSI/MIL-STD-1815A ANNEX OR APPENDIX**

GENERIC INPUT-OUTPUT

DATE: September 30, 1988

NAME: Lee MacLaren

ADDRESS: Boeing 4M-14
P.O. Box 3707
Seattle, WA 98124

TELEPHONE: (206) 655-3472
E-mail: via Maretta Holden on the JIAWG bulletin board

ANSI/MIL-STD-1815A REFERENCE: Annex A (02)

PROBLEM:

The ability to treat any of a wide variety of objects as an array of storage units is required in most embedded systems. Generic input-output is a very common and easy to understand example of this basic requirement.

SPECIFIC REQUIREMENT:

It must be possible for an application programmer to write a service package to perform input-output, or some other generic operation, on objects of arbitrary type. The operation involves treating the objects as blocks of bytes, rather than as objects of the user-declared type. As a test case for meeting this requirement, it should be possible to implement package `Sequential_IO` (ARM 14.2.3) in Ada without recourse to compiler-specific tricks. (Clearly, the body of `Sequential_IO` will be specific to the target computer and operating system. However, it would be nice to be able to survive new releases of the compiler or even rehosting to an alternate compiler.)

Two additional requirements arise in embedded systems:

- Extra copying to and from user buffers must be avoided in order to conserve both space and time. Thus it is expected that the data bytes (or whatever) will be copied directly between the user's object and the external device without intermediate buffering.
- Input-output transactions occur in parallel with user processing. The input-output system must therefore be given the address of the user's object, not just the name of a formal parameter. (This will become more clear in the example below.) It is not appropriate to require that the user construct a pointer to his/her data object. Most such objects will be "statically" allocated in library packages, rather than being allocated on the heap. Instead, an interface package should be responsible for converting an object name into an address and a size.

IMPORTANCE:

A very large fraction of embedded systems encounter the need to treat various objects as blocks of bytes, and a way will always be found to do it. The impact of continuing with current approaches is vulnerability to new compiler releases and difficulty in re-hosting to alternate compilers.

CURRENT WORKAROUNDS:

The obvious (but seriously flawed) approach is to use the 'ADDRESS and 'SIZE attributes, together with Unchecked_Conversion on pointers. For example:

```

package Broadcast_IO is

-- This is the basic interface package for sending or receiving
-- broadcast messages. To use it, the generic package Routines
-- must be instantiated for each message buffer type.
--
-- with Broadcast_IO;
-- with NAV_IO;
-- package user is
--
-- ...
--   package NAV_16 is new
--     Broadcast_IO.Routines (NAV_IO.State_Type);
--   ...
--   NAV_State: NAV_IO.State_Type;
--   ...
--   NAV_16.Connect (NAV_IO.StateA_MsgID, NAV_State);
--
-- WARNING: This package assumes that the Buffer parameters are
-- always passed by reference. If they are not, the low level
-- routines will not be able to find the caller's message buffer.

type MsgID_Type is new Integer;

generic
  type Buffer_Type is private;
package Routines is

  procedure Send (MsgID: MsgID_Type;
                  Buffer: Buffer_Type);
    -- The contents of Buffer will be delivered to all
    -- receiving buffers that have Connected to this MsgID

  ...

end Routines;

end Broadcast_IO;

-----

with Broadcast_Machinery;
package body Broadcast_IO is

package BM renames Broadcast_Machinery;

```

package body Routines **is**

Buffer_Size: **constant** Integer := (Buffer_Type'Size + 7)/8;
 -- The size in bytes of Buffer_Type objects

procedure Send (MsgID: MsgID_Type;
 Buffer: Buffer_Type) **is**
begin

 -- Here is the conversion from the user's buffer object to
 -- an address and a size. Inside Broadcast_Machinery
 -- the address is converted to an access value pointing
 -- to a block of bytes.

BM.Send (BM.MsgID_Type(MsgID), Buffer'Address, Buffer_Size);
end Send;

...

end Routines;

end Broadcast_IO;

with System; **use** System;

package Broadcast_Machinery **is**

type MsgID_Type **is** ...

procedure Send (MsgID: MsgID_Type;
 BufAddr: Address;
 Size: Integer);
 -- Copy from BufAddr to all buffers connected to MsgID

 ...

end Broadcast_Machinery;

with Unchecked_Conversion;
package body Broadcast_Machinery **is**

MaxMsgSize: **constant** := 10_000; -- bytes

```

...

type Byte is range -(2**7)..(2**7)-1;
  for Byte'size use 8;

type MsgBuffer_Type is
  array (Integer range 1 .. MaxMsgSize) of Byte;

type MsgPtr_Type is access MsgBuffer_Type;

function Address_to_Pointer is new
  Unchecked_Conversion (Address, MsgPtr_Type);
function Pointer_to_Address is new
  Unchecked_Conversion (MsgPtr_Type, Address);

...

procedure Send      (MsgID:      MsgID_Type;
                    BufAddr:    Address;
                    Size:       Integer) is
-- Put a copy of the message into each connected buffer

From:  MsgPtr_Type := null;
To:    MsgPtr_Type := null;
XferSize: Integer := 0;
...

begin
  From := Address_to_Pointer(BufAddr);
  loop
    -- step through each entry connected to MsgID
    ...
    To := Address_to_Pointer( ... rcv buffer addr ...);
    XferSize := ... rcv buffer size ...

    if Size < XferSize then
      XferSize := Size; -- don't send more than there is
    end if;
    To(1..XferSize) := From(1..XferSize);
    ...
  end loop;
  ...
end Send;

...

end Broadcast_Machinery;

```

The following flaws in this approach have been observed (there may be more!):

1. It requires that Buffer'ADDRESS be the address of the buffer itself, and not a separate dope vector or some such thing. Buffer must be a contiguous object.

2. Buffer_Type'SIZE must be the size of the actual Buffer objects. A variant record type with default discriminant values will have a 'SIZE large enough for any value of the type, but an object of that type allocated on the heap may be smaller. Since the formal parameter is unconstrained, it is not clear whether applying 'SIZE to the parameter instead of the type will help or not.
3. It is not guaranteed that the representation of System.Address will be the same as that for pointers (access objects). While this author is not aware of any Ada compilers that do not use simple addresses for pointers, this objection is often raised.
4. The approach requires that the Buffer parameter not be passed by value. This is not difficult to achieve in practice, but it seems unclean. There does not seem to be a way to detect a violation of this assumption inside the package.

POSSIBLE SOLUTIONS:

This is a rather dirty corner of "real" programming, and no clean solutions are immediately obvious.

It should, as a minimum, be possible to test attributes of generic actual parameters and to generate compile-time errors if the required properties are not present. If compile-time errors can't be arranged, there are always run-time exceptions. One can imagine attributes such as 'CONTIGUOUS, 'FIXED_SIZE, and 'REFERENCE_PARAMETER.

It is probably too much to ask that non-contiguous objects be prohibited. But if they are going to be allowed, we need a way to deal with the parts.

Another aspect of the problem is guaranteeing fixed and appropriate sizes for objects, especially for variant records. A constant source of difficulty for embedded system programmers is the escape clause (13.2(5)) that permits a compiler to allocate less than 'SIZE bits for an object, even when an explicit length clause is employed. This adversely affects not only the ability to do input-output but also the ability to do assignments to heap objects and to control interfaces with external devices. A mechanism such as a special form of length clause for guaranteeing fixed size objects would be helpful.

It might be appropriate to provide explicit functions for constructing pointers from addresses and vice versa.

DIFFICULTIES TO BE CONSIDERED:

See the list of flaws at the end of the justification section.

SECONDARY STANDARDS FOR PREDEFINED LIBRARY UNITS (ADA-UK/006)

DATE: March 23, 1989

NAME: John Dawes (from material prepared by members of Ada UK)

ADDRESS: ICL
Eskdale Road
Winnersh
Wokingham
Berkshire RG11 5TT
UK

TELEPHONE: +44734693131

ANSI/MIL-STD 1815A REFERENCE: Annex C

PROBLEM:

The standard does not truly reflect the actual situation with regard to standard library packages; for instance implementations for embedded systems need not provide TEXT_IO to be validated; in this respect TEXT_IO is more like e.g., a mathematical library package. This makes the border between what is part of the language and what is not unnecessarily fuzzy, giving problems to both implementors and users, and making the status of validation unclear.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

Implementors supply what seems reasonable to them and the Validation Facility.

POSSIBLE SOLUTIONS:

The definitions of the input-output packages (and some others) should be removed from the language standard and made into secondary or incremental standards, in the same way as is proposed for mathematical functions; that is, the packages need not be provided, but if they are provided they must satisfy the standard. An implementation may provide other forms of textual input-output (e.g., for an interactive terminal) etc., but may not use the names of the predefined library units.

It is not exactly clear where the boundary should be drawn between library units that should remain in the primary standard and those that should be taken out as secondary standards. The following is proposed as a basis for discussion.

"Secondary standards for Ada high-level input-output: IO_EXCEPTIONS, SEQUENTIAL_IO, DIRECT_IO, TEXT_IO.

Reason: They cannot be implemented, and are not needed, on embedded targets. They are presented as four separate standards for flexibility, so that when appropriate a subset can be provided, and so that a nonstandard input-output package can be written using IO_EXCEPTIONS.

"Secondary standard for Ada low-level input-output: `LOW_LEVEL_IO`.

Reason: This is not needed and cannot always be usefully implemented on some targets, e.g., mainframes with modern operating systems.

"Secondary standard for Ada machine code subprograms: `MACHINE_CODE`.

Reason: This is already optional.

"To remain in the primary standard: `STANDARD`, `SYSTEM`, `UNCHECKED_DEALLOCATION`, `UNCHECKED_CONVERSION`, `CALENDAR`.

Reason: `STANDARD` contains the declarations of the predefined types and other language-defined entities.

- `SYSTEM` contains several declarations of entities forming part of the language, including type `ADDRESS` and subtype `PRIORITY`.
- `UNCHECKED_DEALLOCATION` and `UNCHECKED_CONVERSION` are really language features presented as library units, and as such are usually implemented implicitly by a compiler.
- `CALENDAR` might be thought to be a borderline case if one considers implementation on a processor without a real-time clock, but it is considered to be part of the language as access to the clock is essential for certain classes of applications, e.g., for benchmarking and time-stamping.

USE FULL EXTENDED ASCII CHARACTER SET**DATE:** February 20, 1989**NAME:** Chris Clarke**ADDRESS:** Berkshire Fox & Associates
Post Office Box 90673
Pasadena, CA 91109-0673**TELEPHONE:****ANSI/MIL-STD-1815A REFERENCE:** Annex C, Page C-3, Paragraph 13**PROBLEM:**

The original Ada language specification was prepared at a time when most computer applications were character-based. In contrast, today, more and more users are demanding graphics-based applications. Business application programmers are increasingly using the graphics symbols in the Extended ASCII character set to construct graphic images (ie., dialogue boxes, pull-down menus, etc.) in their programs. The current version of Ada only provides access to the original 128 ASCII symbols through the predefined type CHARACTER within the package STANDARD. In order to gain access to the Extended ASCII graphic symbols, programmers must resort to using non-standard, proprietary graphics packages which are supplied by compiler manufacturers. This can severely reduce the portability of their programs.

IMPORTANCE: ADMINISTRATIVE

If this request is not satisfied, then, Ada programs with graphic images will continue to be nonportable.

CURRENT WORKAROUNDS:

Many compiler manufacturers include non-standard, proprietary packages with their compilers for creating graphic images.

POSSIBLE SOLUTIONS:

Expand the predefined type CHARACTER to include all of the 256 symbols in the Extended ASCII character set. In addition, expand the package ASCII to include names for the Extended ASCII symbols.

REQUIRED VENDOR DOCUMENTATION**DATE:** July 13, 1989**NAME:** M. Ben-Ari**ADDRESS:** Brandeis University (until 8/89)81 Hagedud Haivri St. (from 9/89)
Kiryat Haim 26306 Israel**TELEPHONE:** 617-736-2726 or 617-332-1419 (until 8/89)
011-972-4-725905 (from 9/89)
E-mail: moti@cs.brandeis.edu (until 8/89)**ANSI/MIL-STD-1815A REFERENCE:** Appendix F**PROBLEM:**

If the Ada language standard is not to be overspecified, it is inevitable that embedded systems software will be partially implementation dependent. Those aspects of the Ada language that are not specified by the standard must be documented in a comprehensive and consistent manner. This is essential so that software engineers can compare implementations and can document those parts of the systems that are implementation dependent. While many vendors do in fact supply good documentation of the machine interface, the information is not always complete and is certainly not consistent among vendors.

IMPORTANCE: ADMINISTRATIVE**CURRENT WORKAROUNDS:**

The only alternative is to depend on the goodwill of the vendor or to obtain the information by deciphering machine code.

POSSIBLE SOLUTIONS:

Appendix F be expanded to list all permitted deviations from the standard and all aspects of the languages that are implementation dependent. Conforming implementations will be required to supply the information list in the appendix.

The following is a preliminary listing of information required by a software engineer working on embedded systems:

1. Representation of objects
 - Enumeration types
 - Composite objects
 - Dope Vectors
 - Variant records
2. Memory allocation scheme
 - Linker interface
 - Stack and heap use
 - Unconstrained records

- 3. Subprogram calling mechanism
 - Parameter passing mechanism
 - pragma Interface
- 4. Tasking
 - Time-slicing
 - Choice of alternative in select statement
 - Synchronous or asynchronous abort
- 5. Exceptions
 - Hardware traps
 - Software traps
 - pragma Suppress

**16. REVISION REQUESTS THAT DO NOT
REFERENCE ANSI/MIL-STD-1815A**

LEAVE UNSIGNED NUMBERS OUT**DATE:** February 21, 1989**NAME:** SY Wong**ADDRESS:** 5200 Topeka Drive
Tarzana, CA 91356**TELEPHONE:** (818) 345-6274
E-mail: hermix!sywong@rand-unix.arpa**ANSI/MIL-STD-1815a REFERENCE:****PROBLEM:**

Addressing and indexing calculations traditionally use 0+ positive numbers. This problem only became critical on some 16-bit machines where physical address space is the same as the data word. Most older and more modern machines with larger word lengths that exceeds most memory address space do not need unsigned arithmetics. Modern memory mapping and management chips further obviate the need. Unsigned arithmetics just clutter computer design unnecessarily. More appropriate is to alert hardware designers to use a uniform mapping of numbering scheme for address and numbers.

Random number generators requires the inhibition of overflow and is independent of unsigned number consideration.

IMPORTANCE: ESSENTIAL**CURRENT WORKAROUNDS:****POSSIBLE SOLUTIONS:**

Leave unsigned number out in Ada and leave the subject to the Ada and architecture issue committee.

EXTEND USE OF "&", "|" AND KEYWORD "IS" AND "NOT IS"

DATE: February 22, 1989

NAME: SY Wong

ADDRESS: 5200 Topeka Drive
Tarzana, CA 91356

TELEPHONE: (818) 345-6274
E-mail: hermix!sywong@rand-unix.arpa

ANSI/MIL-STD-1815A REFERENCE:**PROBLEM:**

Not too readable and tedious is

1. if long_expression = A or else long_expression = B or else long_expression = C then
2. Definition of sets via several subtypes or boolean arrays.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS: No direct and convenient ways.

POSSIBLE SOLUTIONS:

Extend the use of '|' and keyword is and not is

1. becomes:

if long_expression is A | B | C then
if long_expression not is A | B | C then

This also avoids the use of the use clause in using infix notation. Extend the use of '&', '|' and keyword is and not in.

2. subtype letters is character range 'A'..'Z' & 'a'..'z';
subtype control_characters is character range
ascii.nul..ascii.us & ascii.del..ascii.del;
subtype symbols is character not in letter
control_characters;
subtype graphical_characters is character in symbols letters;

DEFINITION AND USE OF TECHNICAL TERMS ADA-UK/004

DATE: March 23, 1989

NAME: John Dawes (from material prepared by members of Ada UK)

ADDRESS: ICL
Eskdale Road
Winnersh
Wokingham
Berkshire RG11 5TT
UK

TELEPHONE: +44734693131

ANSI/MIL-STD 1815A REFERENCE:

PROBLEM:

The use of terms in the Reference Manual is not always consistent, and not all terms are well defined. The result is that the Reference Manual is needlessly hard to understand, wasting the time of implementors, programmers, educators, and many others who are driven to rely on ancillary sources such as Ada commentaries.

Some notes on particular cases are appended to show the sort of problem that is referred to here. These are intended as examples only; many more similar cases could be cited.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

Ada commentaries are a necessary supplement to the reference manual, and in some cases the ACVC suite becomes the de facto language definition. Neither of these is satisfactory.

POSSIBLE SOLUTIONS:

Many instances of the problem have been identified as Ada Issues and are being addressed; however the problem needs to be attacked systematically rather than piecemeal, or there is a danger of introducing new inconsistencies while clearing up old ones. Great care will be needed if the effort to improve precision is not to lead to a loss of readability.

A possible programme to address the problem is:

- (1) Make a definitive collection of all technical terms.
- (2) Ensure that a definition of each technical term exists in an identifiable place in the text of the Reference Manual.
- (3) Ensure that each use of a technical term in the reference manual is consistent with its definition.

1. Terms - Definition and Use.

The Reference Manual (RM) suffers from lack of an explicit conceptual model against which to define the semantics of the language, and as a result has difficulty in defining terms in a way that stands up to analysis. This leads to inconsistent use of terms which in turn makes it difficult for the reader to understand the intention. In this paper I make some general remarks about the definition and use of terms in the RM and then analyses two examples chosen more or less at random.

1.5(2,3) The meaning of Ada program units is described by means of narrative rules defining both the effects of each construct and the composition rules for constructs. This narrative employs technical terms whose precise definition is given in the text (references to the section containing the definition of a technical term appear at the end of each section that uses the term).

All other terms are in the English language and bear their natural meaning, as defined in Webster's Third New International Dictionary of the English Language.

Let us forgive the wishful thinking of 'precise'. Evidently any term is either technical or not, i.e., no term is used sometimes technically and sometimes naturally. How can we tell whether a term is technical or not? 1.5(2) might be read as saying that technical terms are just those for which references appear at the ends of sections (despite the fact that the references are not actually part of the standard). This is supported by the observation that in many cases the section referred to contains an italicized occurrence of the term, which may indicate that the sentence or clause containing it is the 'precise definition' referred to above. Another criterion, which appears to define the same set of technical terms, is given in the preamble to the Index:

An entry exists in this index for each technical term or phrase that is defined in the reference manual. The term or phrase is in boldface and is followed by the section number where it is defined, also in boldface, for example:

Record aggregate 4.3.1

Having decided that a term is technical, how can we find its definition? 1.5(2) and the Index narrow the search to one section for each term; but we are then reduced to searching the text of the section for a likely looking form of words. Italicisation is a fallible guide - it may just indicate the first occurrence of the term in the RM. For instance I don't see how anyone could read the following as a definition of 'formal parameter':

6.1(3) The specification of a procedure specifies its identifier and its formal parameters (if any)... And not all technical terms (on the above criterion) have italicized occurrences; as it happens both my examples include cases of this sort.

2. Types and Subtypes

Consider the following.

1.4(20) Every object in the language has a type, which characterizes a set of values and a set of applicable operations.

3.3(1) A type is characterized by a set of values and a set of operations.

3.3(4) The set of possible values for an object of a given type can be subjected to a condition that is

called a constraint (the case where the constraint imposes no restriction is also included); a value is said to satisfy a constraint if it satisfies the corresponding condition. A subtype is a type together with a constraint; a value is said to belong to a subtype of a given type if it belongs to the type and satisfies the constraint; the given type is called the base type of the subtype. A type is a subtype of itself; such a subtype is said to be unconstrained: it corresponds to a condition that imposes no restriction. The base type of a type is the type itself.

3.3.1(5) The simple name declared by a full type declaration denotes the declared type, unless the type declaration declares both a base type and a subtype of the base type, in which case the simple name denotes the subtype, and the base type is anonymous.

3.3.2(3) A type mark denotes a type or a subtype. If a type mark is the name of a type, the type mark denotes this type and also the corresponding unconstrained subtype.

The following difficulties occur:

- Although 1.4(20) is not part of the standard, there seems to be some doubt as to whether the type characterizes the sets of values and of operations or vice versa. I confess that I cannot make much sense of either sentence. 'Characterize' is not a technical term so perhaps Webster could clear this up, but I do not have it to hand.
- The above is not a mere quibble as 3.3(1) seems to be meant to be the definition of the technical term 'type' (though the word is not italicized therein). At any rate I cannot find anything else in the RM that would serve as its definition.
- Is a constraint a condition, or does it merely correspond to one? 3.3(4) says plainly enough that a constraint is a condition, but then goes on to give a definition of 'satisfies' (for a constraint) that only makes sense if a constraint is not a condition.
- The term being introduced or defined in the second clause of the second sentence of 3.3(4) is not 'belong to a subtype', it is 'belong to' referring to a subtype (otherwise the reference to 'the constraint' later in the clause does not refer to anything. The clause should be:

a value is said to belong to a subtype of a given type if it belongs to the type and satisfies the constraint;

- Is a type a subtype? 3.3(4) implies that it is not (since a subtype is a type plus a constraint, which is not the same thing as a type), but also says plainly that it is ('a type is a subtype of itself'). 3.3.1(5) makes no sense if a type is a subtype of the base type', namely the type itself. 3.3.2(3) tries to hedge by using the word 'corresponding' (again!).
- If a subtype indication includes an explicit constraint that imposes no restriction, is the subtype it defines constrained or unconstrained? 3.3(4), third sentence seems to imply that it is constrained, and this is consistent with other parts of the RM, e.g., 3.6.1(6). However, this subtype has the same base type and constraint (or at least the same condition) as the type, considered as a subtype; so how does it differ?
- What is an unconstrained type? There are a number of references to the term 'unconstrained type' (e.g., 3.7.4(3)), and even an index entry (to 3.3); but nowhere in 3.3 or elsewhere can I find a definition of it. It seems to be just a would-be helpful synonym for 'type'.

This could all be cleared up by some careful analysis and rephrasing. I suggest something along the following lines.

- A type is a set of values and a set of operations applicable to those values (i.e., the implicitly declared operations for the type). Some types are predefined by the language; others are created by elaboration of type declarations. Each value belong to just one type.
- A subtype of a type is a subset (possibly empty or the whole type) of the values of that type, which is the base type of each of its subtypes.
- A constraint is a syntactic construct whose elaboration yields a condition on the values of a type which defines a subtype of that type. There is no need to talk of (un)constrained types and subtypes, since types are always unconstrained and subtypes always constrained.

3. Programs and Program Units

The lack of a consistent conceptual model is particularly noticeable in the area of program structured.

'Program' is a technical term defined as follows:

- 10(1) A program is a collection of one or more compilation units submitted to a compiler in one or more compilations.

A program is therefore a collection of source text; however the very next paragraph contradicts this:

- 10.1(1) The text of a program can be submitted to the compiler in one or more compilations.

So a program is something other than its text. The Glossary doesn't help:

- D Program. A program is composed of a number of compilation units, one of which is a subprogram called the main program.

Execution of the program consists of....

This cannot be right, as a subprogram is not a compilation unit (see 10.1). Perhaps 'compilation unit' here is a mistake for 'program unit', which leads us to ask what exactly is a program unit. The index refers us to four sections for its definitions; the relevant paragraphs are:

- 6(1) Subprograms are one of the four forms of program unit, of which programs can be composed. The other forms are packages, task units, and generic units.
- 6(2) ... The definition of a subprogram can be given in two parts: a subprogram declaration defining its calling conventions, and a subprogram body defining its execution.
- 7(1) Packages are one of the four forms of program unit, of which programs can be composed. The other forms are subprograms, task units, and generic units.
- 7.1(1) A package is generally provided in two parts: a package specification and a package body...
- 9(4) The properties of each task are defined by a corresponding task unit which consists of a task specification and a task body. Task units are one of the four forms of program unit, of which programs can be composed. The other forms are subprograms, packages and generic units....
- 12(1) A generic unit is a program unit that is either a generic subprogram or a generic package. A generic unit is a template, which is parameterized or not, and from which corresponding

(nongeneric) subprograms or packages can be obtained. The resulting program units are said to be instances of the original generic unit.

12(2) A generic declaration declares a generic unit, ...

I find it very difficult to gather a clear definition of a program unit from this - it seems to be a vaguely imagined entity which is somehow represented by, but is different to, some source text.

The same confusion occurs with 'library unit' (does it include preceding pragmas or not?) and 'program library'.

The solution is to decide on a conceptual model for the language definition, and to define terms carefully in the light of the model. It is possible to define the semantics of a language entirely in terms of the source text, with no concept of compilation, or in terms of conceptual compilation and execution; it is not possible consistently to switch back and forth between the two, or to rely on a common understanding of the underlying model.

4. Other Examples

Some other examples which I would analyse if I had the time:

- The ontological problem: when are entities created? Sometimes the RM speaks of the elaboration of a declaration creating an entity, and at other times it seems to deliberately avoid doing so.
- What is a 'predefined operation'? This term is given in the index, and used (inconsistently) in several places, but does not appear to be defined anywhere.
- Is a formal parameter a name or an entity? In section 6.2, where valiant efforts are being made in AI-00178 to clarify the semantics of parameter passing, the term 'formal parameter' is used almost always to denote an object, not its name (an exception is 6.2(16)). Unfortunately a formal parameter is in fact a name, since `formal_parameter` is a syntactic category equal to `simple_name` (see 6.4(2)). The recommendation in AI-00178 follows this usage, which is correct but makes it hard to relate it to the RM. The sets of terms for syntactic categories and semantic entities should be disjoint.

APPLICABILITY TO DISTRIBUTE SYSTEMS (ADA-UK/007)

DATE: March 28, 1989

NAME: JGP Barnes (from material supplied by members of Ada-UK)

ADDRESS: Alslys Ltd
Newtown Road
Henley-on-Thames
Oxon
RG9 1EN
UK

TELEPHONE: +44-491-579090

ANSI/MIL-STD 1815A REFERENCE:

PROBLEM:

The Ada language contains a number of features which are obstacles to the use of Ada for generating a complete single program across a distributed set of processors. This is particularly acute when the processors are not all of the same architecture. Examples are:

There is a single package SYSTEM for the whole program and this implies that all those items in SYSTEM are common to all parts of the hardware running the program. This includes STORAGE_UNIT, the number of bits in a storage unit.

There is an implication that there is a single simultaneous CLOCK across the whole system and a fixed granularity to DURATION. Apart from the academic conflict with special relativity, the package CALENDAR may not be directly available throughout the system. The problem of simultaneity raises questions regarding the meaning of timed and conditional calls.

At the present time the problems of distributing a complete program across an arbitrary system remain unsolved. However, it seems unwise to leave unnecessary obstacles to progress in the way if they can be removed.

The advantage of describing a distributed program as a single program is that full checking is obtained across component boundaries.

It is therefore suggested that the Ada 9x revision should include a thorough survey of the language with a view to reconsidering facilities which look like being an obstacle to multiprocessor distributed systems.

IMPORTANCE: ESSENTIAL for distributed systems.

The direct applicability of Ada to multiprocessor distributed systems is impaired.

CURRENT WORKAROUNDS:

Systems have to be "written" as several distinct programs especially if different architectures are involved. Ad hoc linkage schemes then have to be devised.

POSSIBLE SOLUTIONS:

Work on units distribution (Zones) for Ada has been in progress for some time. This has identified constraints on the use of certain facilities across zones (e.g., entry calls with access parameters). Such Zones might have their own copies of packages such as SYSTEM so that different parts of the total program could be on different architectures.

It should be noted that it could be the case that a single compiler is capable of generating code for different architectures (e.g., different models of the same range such as transputer T400 and T800 which have the same order code but different attributes).

Although the total problem looks very hard to solve right now, nevertheless some small changes might be made which lay a foundation for the future.

REFERENCES:

Ada for Distributed Systems, Atkinson C, Moreton T and Natali A, Cambridge University Press, 1988.

RUN-TIME ENVIRONMENT DEFINITION AND INTERFACE

DATE: March 20, 1989

NAME: M.J. Pickett (from material supplied by members of Ada UK)

ADDRESS: Sema Group plc
Orion Court
Kenavon Drive
Reading
Berkshire
RG1 3DQ
UK

TELEPHONE: +44 734 508961

ANSI/MIL-STD-1815A REFERENCE:

PROBLEM:

The standard fails to define any kind of interface to a run-time support environment. Ada is a language designed for use in embedded systems, and at the time the language was designed, the microprocessors in such systems had a straightforward architecture and not operating system. Modern processors for embedded systems are substantially more complex and are often required supported by special purpose real-time kernels. The effort required to construct an efficient Ada run-time system for these modern architectures is considerable and duplicates work already being put in to develop the real-time kernels. However, the lack of standardization of Ada in this area effectively prevents the use of these kernels in well defined ways. Thus, there is no defined interface beneath an Ada program, and consequently Ada programs are not portable across different real-time kernels.

IMPORTANCE: ESSENTIAL

For system developers in the field of real-time embedded systems, the efficiency of the run-time system is vital. This usually results in the developer evolving his own real-time kernel, either in house or in conjunction with a specialist supplier, over a period of years. In either case, the developer expects to be able to tune the kernel to the application, and he has the experience to do this. In general, the run-time system supplied for an Ada compiler is general purpose and has only limited tuning facilities.

Ada has received a lot of bad publicity for the poor performance of its programs. Therefore system developers expect the area of the run-time systems to be open for tuning. If it is not, they will favor other languages to Ada where they have a choice, and Ada will suffer further, unnecessary setbacks.

CURRENT WORKAROUNDS:

There is no real workaround for the user: the choice is between preserving portability by accepting any overheads that may result from using pure Ada with whatever underlying real-time kernel has been chosen, or ignoring the Ada features and making calls directly to the facilities of the real-time kernel. Some suppliers may adapt their products to use particular real-time kernels, but this generally results in adapting the interface to the kernel as well. Thus there is no standardization of the kernel interfaces across different compilers.

POSSIBLE SOLUTIONS:

There are four levels at which standardization might be applied. At the top level, the definition of the language could be extended to address the relevant issues. At a level only slightly lower, there could be a secondary standard defining services and interfaces to be provided for interaction between the run-time system and any underlying real-time kernel. At a level lower still, these services and interfaces could be defined in an optional addendum to the standard. At the bottom level there could be a requirement for the developer of the compilation system to list which of a defined set of services and interfaces he has provided and to specify how they may be supported.

The most attractive of these approaches is the development of a secondary standard in the same time frame as the revision of the standard.

Wherever possible, the secondary standard should define functional interfaces to the run-time system. This would enable real-time kernels to be written to these interfaces, so preserving the Ada semantics whilst giving the application developer the freedom to use his skill and experience.

Particular interfaces should include:-

- Interrupt handlers;
- Scheduling algorithm;
- Storage allocation;
- Garbage collection;
- Context saving;
- Time.

A more thorough examination of the requirements in this area can be found in the work of the ARTEWG, and in particular in the document "Catalogue of Interface Features and Options for the Ada Run-time Environment".

STREAM I/O**DATE:** April 21, 1989**NAME:** John Hedstrom**ADDRESS:** Texas Instruments
P.O. Box 869305 MS 8435
Plano, TX 75086**TELEPHONE:** (214) 575-4124**ANSI/MIL-STD-1815A REFERENCE:****PROBLEM:**

Digital signal processing applications must continuously process streams of data. The Ada language does not presently provide built-in input-output stream primitives such as `STREAM_CREATE`, `STREAM_GET`, and `STREAM_PUT` to manipulate pre-determined amounts of data. Such functions can be synchronized with external devices. Stream input-output primitives could be added to package `STANDARD`.

IMPORTANCE: IMPORTANT**CONSEQUENCES:**

Builders of signal processing applications are not motivated to use Ada (i.e., no natural constructs available to aid them.)

CURRENT WORKAROUNDS:

Must write device specific packages to transfer data to and from the application program.

POSSIBLE SOLUTIONS:

PACKAGE/SUBPROGRAM TYPES

DATE: April 21, 1989

NAME: Stewart French

ADDRESS: Texas Instruments
P.O. Box 869305
MS 8435
Plano, TX 75086

TELEPHONE: (214) 575-3522

ANSI/MIL-STD-1815A REFERENCE:

PROBLEM:

Provide Ada types for packages and subprograms. Extend the ability to define user types; would allow passing of subprograms as parameters.

IMPORTANCE: IMPORTANT

CONSEQUENCES: Limits flexibility of user defined types.

CURRENT WORKAROUNDS:

POSSIBLE SOLUTIONS:

INCONSISTENCY IN ADA SEMANTICS OF RACE CONTROLS**DATE:** June 1, 1989**NAME:** Tzilla Elrad**ADDRESS:** Illinois Institute of Technology
Department of Computer Science
IIT Center
Chicago, IL 60616**TELEPHONE:** (312) 567-5142**ANSI/MIL-STD-1815A REFERENCE:****PROBLEM:**

The full power of Ada tasking model can not be used for real-time applications because of lack of control over scheduling decisions.

IMPORTANCE: ESSENTIAL**CURRENT WORKAROUNDS:**

Nested selects, families of entries, 'count attribute, specific compiler implementation.

POSSIBLE SOLUTIONS:

- A. Whenever a choice is to be made within Ada tasking that affects which task will be eligible to execute, Ada tasks priorities must be used to make the choice.
- B. Whenever a choice is to be made within Ada tasking, the user shall be able to control the choice. This implies:

Program level: Dynamic priority
Task level: Associate priority control to alternative within a select
Entry level: Associate priority control to entry calls

The Second requirement gives more scheduling flexibility than the first. It implies explicit control over the choice of an alternative within a select, and explicit control over the choice of pending calls to an entry. However the choice based on task priorities. Hence the first alternative to specify requirements is automatically satisfied by the second.

The complete set of race controls will serve as a versatile mechanism for the implementation of a wide range of scheduling alternatives that the Ada community is very likely to face in the future. More research is needed to investigate the second alternative.

SUGGESTED STANDARD PACKAGE FOR BIT-LEVEL OPERATIONS ON INTEGER DATA TYPES

DATE: June 23, 1989

NAME: Cliver M. Kellogg

ADDRESS: MBB BmbH
Space Systems Group
Dept. KT 142
P.O. Box 801169
D-8000 Munich 80
FRG

TELEPHONE: (-49) 89 60008125

ANSI/MIL-STD-1815A REFERENCE: NONE

PROBLEM:

Non-portability caused by leaving the implementation of bit operations (especially Shift Left / Right) at the complete freedom of compiler vendors.

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS:

Depend on compiler used. Many vendors furnish at least bitwise AND, OR, and NOT operators on the native integer types of the target machine in a non-standardized manner. ARithmetic workarounds (e.g. multiplication or division by 2 instead of shift operations) are deemed inadequate due to near-overflow problems and compiler implementation dependencies.

POSSIBLE SOLUTIONS:

The vast majority of processors dispose of single machine instructions for bit-level operations such as AND, OR, NOT, EXOR, Shift Right / Left by N bits. The proposal aims at establishing a standard package ("BIT_OPS") for the exploitation of this machine functionality in a compiler independent manner. Arguably, it would be desirable to implement such a package as a generic.

In the case of shift operations, the expected behavior ought to be such that bits "shifted out" of the machine data operand are lost, and bit positions "vacated" by the shift operation are filled with zeros.

Special attention might be needed if the target processor's native integer types are signed. Some processors treat the sign bit differently than other bits in the data operand. One simple way around this is to define the Ada data type used in the bit operations so as to exclude the sign bit (by use of range constraint or representation spec.)

ADD PREDEFINED ISAM PACKAGE**DATE:** February 20, 1989**NAME:** Chris Clarke**ADDRESS:** Berkshire, Fox & Associates
Post Office Box 90673
Pasadena, CA 91109-0673**TELEPHONE:****ANSI/MIL-STD-1815A REFERENCE:** NONE**PROBLEM:**

In business, each piece of information is usually coded with a unique identifier (ie., last name, account number, employee number, vendor number, etc.). When creating computerized information systems for businesses, programmers require the ability to quickly store and access information by reference to this unique identifier (called the record key). One widely used tool for doing this is the Indexed Sequential Access Method (ISAM). The COBOL programming language includes ISAM as a standard feature. Ada does not. As a result, when using Ada for business systems, programmers are forced to design, code and test their own custom ISAM package. The extra expense required to do this creates a powerful disincentive for business programmers to switch from COBOL to Ada.

IMPORTANCE: ADMINISTRATIVE

Because of the extra cost required to create a custom ISAM package, many business programmers will be deterred from switching to Ada.

CURRENT WORKAROUNDS:

A custom ISAM package is either written by the programmer or supplied by the compiler manufacturer.

POSSIBLE SOLUTIONS:

Add a new predefined ISAM package to Ada. At a minimum, this new package should;

- a) include the procedures CREATE, DELETE, START, READ, READ_FOR_UPDATE (which locks the record), READ_NEXT, WRITE, REWRITE (which releases a locked record), and DELETE_RECORD,
- b) have the capability to create indexes for multiple key fields,
- c) support input-output of variable length records, and
- d) detect invalid keys.

In short, it should at least include all of COBOL'S ISAM capabilities.

PROVIDE CHAINING CAPABILITY IN PREDEFINED PROCEDURE**DATE:** February 20, 1989**NAME:** Chris Clarke**ADDRESS:** Berkshire, Fox & Associates
Post Office Box 90673
Pasadena, CA 91109-0673**TELEPHONE:****ANSI/MIL-STD-1815A REFERENCE:** NONE**PROBLEM:**

Due to the availability of virtual memory, most minicomputer and mainframe programmers rarely consider the size of main memory as a limiting factor when creating their programs. In contrast, the size of main memory is a major concern of microcomputer programmers. The most widely used microcomputer operating systems, MS-DOS, does not have virtual memory capabilities. Without the availability of special programming techniques to get around this limitation, microcomputer programmers would have to severely limit the functionality of their programs, and, it would be impossible to create large, integrated information systems for microcomputers. One of most widely used of these programming techniques is the "chaining" capability provided in many programming languages. "Chaining" gives a programmer the ability to break down large integrated information systems into separate executable programs, and, then, when the system is operated, swap these programs in and out of main memory as the need arises. "Chaining", in effect, simulates virtual memory. Ada does not have the capability to chain programs. As a result, microcomputer programmers who use Ada must severely limit the functionality of their programs.

IMPORTANCE: ADMINISTRATIVE

Microcomputer programmers who use Ada will have to continue limiting the functionality of their programs.

CURRENT WORKAROUNDS:

Programmers must either limit the functionality of their Ada programs or use a proprietary CHAIN command supplied by the compiler manufacturer - which hurts portability.

POSSIBLE SOLUTIONS:

Create a new predefined procedure called CHAIN. When invoked, this new procedure would completely transfer control to the named program and deallocate the memory used by the original program. Preferably, the new procedure would also preserve the common data areas in each program.

APPENDIX A.
SAMPLE REVISION REQUEST FORM

January 1989
Revised Form
(Previous version
acceptable)

**Ada 9X
REVISION REQUEST FORMAT**

(Please submit one request per form)

DATE: Provide date that the request is prepared.

NAME: Provide the name of the individual who has prepared the request.

ADDRESS: Provide the name of the individual's organization and mailing address, outside the United States; please include the appropriate country and city codes.

TELEPHONE: Provide telephone number of individual. If request is from outside the United States, please include the appropriate country and city codes.

ANSI/MIL-STD-1815A REFERENCE (Section, Paragraph Number): If no chapter is particularly relevant, please so state.

TITLE: Provide a short title or key words characterizing the request.

PROBLEM: Briefly state the problem. Indicate the most critical aspects to be kept in mind for arriving at a solution, particularly if only a partial solution is possible.

IMPORTANCE: Choose one of the following ratings to describe the importance of the request with respect to the overall Ada community, and discuss the consequences if the request is not satisfied by the revision.

ESSENTIAL: The revised standard is unlikely to be accepted if this revision request is not supported.

IMPORTANT: Nice to have but not essential. Should be supported if minimum negative impact to implementations.

ADMINISTRATIVE: Technical correction that makes the standard more consistent with the design intent or less subject to misunderstanding.

CURRENT WORKAROUNDS: Provide specific examples of workarounds currently being used that allow partial solution to the problem.

POSSIBLE SOLUTIONS (Optional): Discuss possible solutions for addressing the stated problem.

Submit completed form to:

The Ada Joint Program Office
ATTN: Ada 9X Project
Room 3E113, Pentagon
Washington, DC 20301-3080

INDICES

KEY TECHNICAL TERMS

=

1-3, 2-3, 2-5, 2-6, 3-5, 3-6, 3-11, 3-12, 3-15 to 3-21, 3-23, 3-24, 3-28, 3-29, 3-31, 3-35, 3-36, 3-41, 3-42, 3-45, 4-2 to 4-4, 4-6, 4-8, 4-11, 4-14, 4-20, 4-24, 5-4, 5-6, 6-4 to 6-6, 6-8, 6-10, 6-14, 6-15, 6-17, 6-20 to 6-22, 7-8, 7-10 to 7-12, 8-2, 8-5, 8-9, 9-7, 9-8, 9-16, 9-17, 9-32, 9-33, 10-13, 10-17, 11-2, 11-5 to 11-7, 11-16, 13-5, 13-8, 13-21, 13-24, 13-27, 14-6, 14-8, 15-4, 15-5, 16-3

7-bit ASCII

2-2, 8-8

8-bit ASCII

2-2

abort

9-5, 9-35, 9-52, 9-53, 11-3, 13-24, 15-11

abstract data types

7-10

abstraction

6-18, 9-2, 9-19

accept

3-10, 3-30, 9-2, 9-5, 9-10, 9-23 to 9-26, 9-28, 9-31, 9-35 to 9-37, 9-43, 12-2, 13-17

access types

3-4, 3-6, 3-19, 3-38, 3-45, 4-16, 13-24, 13-27

activation

4-17, 9-15-17, 9-39, 11-9

address clause

13-14, 13-15

address

1-2 to 1-5, 2-2 to 2-4, 3-2, 3-5, 3-7, 3-10, 3-14, 3-16, 3-19, 3-20, 3-22, 3-24, 3-25, 3-27 to 3-31, 3-33 to 3-35, 3-37, 3-38, 3-40, 3-41, 3-42, 3-45, 4-2, 4-4, 4-6 to 4-9, 4-11, 4-13 to 4-15, 4-19, 4-22 to 4-24, 5-2, 5-4, 5-6, 5-8, 6-2-4, 6-7, 6-8, 6-10, 6-12 to 6-16, 6-18, 6-20, 6-22, 7-2 to 7-5, 7-7, 7-9, 7-10, 7-12, 8-2, 8-4, 8-6, 8-8, 9-2, 9-4, 9-7, 9-9, 9-12 to 9-15, 9-19 to 9-23, 9-25, 9-28, 9-30, 9-32, 9-35, 9-37 to 9-39, 9-41, 10-2, 10-5, 10-8, 10-10, 10-12, 10-14 to 10-16, 10-18, 11-2, 11-3, 11-5, 11-8, 11-10, 11-11, 11-13 to 11-16, 12-2, 12-3, 13-2 to 13-4, 13-6, 13-8, 13-10 to 13-12, 13-14 to 13-16, 13-18, 13-20, 13-22, 13-23, 13-26 to 13-28, 14-2 to 14-8, 15-2 to 15-10, 16-2 to 16-4, 16-9, 16-11 to 16-18

adjusting

13-20, 13-21

aggregate

1-4, 2-6, 3-2 to 3-4, 4-2, 4-3, 7-13, 16-5

allocators

1-3, 1-4, 3-4, 3-36, 4-16, 7-13, 11-11, 13-23, 13-25

apply

3-11, 3-43, 5-6, 8-5, 9-9, 9-42, 9-54, 10-12, 10-18, 13-8, 13-21

assembly language

4-9, 6-12, 6-14, 9-4, 13-22, 14-5

assignment statement

3-19, 5-4, 5-6, 5-7

asynchronous

9-9 to 9-11, 9-35, 9-36, 9-54

KEY TECHNICAL TERMS

attribute

3-36, 3-37, 3-42, 4-6, 4-19, 4-21, 4-22, 6-3, 9-51,
11-2, 11-4, 13-6, 13-7, 16-15

body

3-4, 3-8, 3-35, 3-47, 4-24, 6-6, 6-10, 7-2, 7-5 to
7-8, 7-11, 8-4, 9-8, 9-15 to 9-17, 9-25, 9-26, 9-32,
9-33, 9-35, 10-5, 10-6, 10-10, 10-12 to 10-14, 10-
17, 11-13, 12-3, 12-4, 13-15, 13-17, 13-22, 15-2 to
15-4

calendar.clock

8-2, 9-30 to 9-33, 13-20, 13-29

case statement

3-30, 3-40, 11-3

chain programs

16-18

cleanup

3-22, 3-23, 9-23, 10-10, 10-11

clock

8-2, 9-28 to 9-34, 13-20, 13-21, 13-29, 15-8, 16-9

compilation units

2-5, 7-2, 10-2, 10-3, 16-7

conditional entry calls

9-12, 9-13, 13-17

constraint_error

3-29, 3-31, 3-32, 4-3, 4-11, 5-6, 9-7, 11-7, 11-10

context clauses

10-4, 10-9, 10-12, 10-13

continuous range

4-8

count attribute

9-51, 16-5

deadlock states

9-52

declaration

1-4, 3-1, 3-4, 3-6, 3-10-14, 3-17, 3-19, 3-20, 3-23,
3-27, 3-30, 3-38, 3-43, 3-45, 3-46, 4-7, 4-16, 4-20,
6-7, 6-17, 6-22, 7-4, 7-6 to 7-8, 7-11, 8-5, 8-6, 9-
16, 9-17, 9-34, 9-55, 11-6, 11-13, 11-14, 13-14, 16-
6 to 16-8

declarative

3-4, 3-5, 3-8, 3-43, 7-5, 7-6, 10-2, 10-4

default

3-14, 3-15, 3-19, 6-12, 6-16, 6-17, 6-19, 9-23, 9-
52, 13-8, 13-9, 13-15, 14-7, 14-8, 15-6

delay

3-32, 9-13, 9-15, 9-28, 9-32 to 9-34, 9-36, 10-3, 10-
10

dependent types

3-45

derivation

3-25, 7-9, 9-19

derived type

7-9, 8-4, 8-6

direct_io

3-14, 3-45, 14-5, 15-7

KEY TECHNICAL TERMS

distributed

5-2, 5-3, 6-2, 6-3, 9-10, 9-12 to 9-14, 9-24, 9-50, 11-9, 13-6, 13-29, 16-9, 16-10

embedded

3-20, 3-33, 3-38, 3-39, 4-18, 5-2, 6-3, 6-12 to 6-14, 9-4, 9-12, 9-14, 9-22 to 9-24, 9-28, 9-30, 9-34, 9-43, 9-50, 10-16, 11-11, 13-12, 13-13, 13-18, 13-20, 13-21, 13-23, 15-2, 15-6, 15-7, 15-10, 16-11

enumeration

3-20, 3-21, 4-6, 4-22, 8-4, 8-6, 8-7, 11-4, 13-3, 13-6 to 13-9, 14-7, 15-10

erroneous execution

1-5, 1-6

error detection

5-2

error recovery

5-2

errors

1-3, 3-16, 3-24, 4-11, 9-30, 10-3, 10-14, 12-3, 15-6

exception

1-5, 3-4, 3-8, 3-10, 3-16-18, 3-28, 3-29, 3-31, 4-9, 4-11, 4-13, 5-2, 6-6, 8-9, 9-10, 9-52, 11-2 to 11-8, 11-10 to 11-17, 13-21, 13-24, 13-27, 16-8

execution

1-3 to 1-6, 3-2, 3-4, 3-40, 4-15 to 4-17, 5-2, 5-8, 6-13 to 6-15, 9-9, 9-10, 9-13 to 9-16, 9-21, 9-28, 9-32, 9-33, 9-36, 9-39, 9-41, 9-45, 9-46, 9-48, 9-49, 9-55, 11-9, 11-11, 11-12, 11-14, 13-16, 16-7, 16-8

exit

3-7, 5-8, 9-21, 9-26, 10-10, 11-15, 11-16, 13-29

exponent/exponentiation

2-3, 4-4, 4-13, 13-29

fault tolerance

5-2, 6-3, 9-13, 9-56

finalization

3-7-9, 3-22, 3-23, 7-14, 8-10, 10-10, 10-11, 10-20

floating point

3-30, 3-33, 3-34, 4-4

function

1-3, 1-5, 2-6, 3-25, 3-39 to 3-43, 4-2, 4-4, 4-11, 4-17, 4-18, 5-8, 5-9, 6-3, 6-5, 6-6, 6-10, 6-14, 6-18 to 6-22, 7-7 to 7-9, 7-12, 8-7, 9-7, 9-17, 9-36, 9-48, 9-49, 9-54, 11-4, 11-10, 13-7, 13-24, 13-26 to 13-28, 14-4, 15-5

garbage collection

3-22, 4-15 to 4-18, 7-3, 13-23, 16-12

generic

2-6, 3-4, 3-10, 3-37, 3-41, 3-43, 3-45, 3-46, 4-2, 4-18, 4-23 to 4-25, 6-2 to 6-4, 6-10, 6-20, 7-9, 7-11, 7-13, 10-10, 10-11, 11-5, 11-7, 11-13, 11-14, 12-1 to 12-4, 13-14, 13-24, 13-28, 14-2, 15-2, 15-3, 15-6, 16-7, 16-8, 16-16

graphics

14-3, 15-9

identifier

2-5, 2-6, 3-5, 3-6, 3-10, 3-12, 3-13, 3-17, 3-19, 4-14, 6-5, 6-6, 6-23, 7-8, 7-14, 8-4, 8-5, 9-15, 9-16, 11-14, 16-5, 16-17

KEY TECHNICAL TERMS

image

3-31, 6-14, 11-2, 11-4, 14-6

implementation dependencies

3-29, 16-16

incomplete

3-11, 3-45, 3-46, 7-7, 7-9, 11-6

independent

4-4, 4-11, 6-2, 9-10, 10-10, 16-2, 16-16

infix notation

3-24, 16-3

information hiding

7-5, 7-6

inheritance

3-5, 3-6, 7-14, 9-5, 9-6, 9-27, 9-46

initialization

3-7, 3-8, 3-12, 3-14, 3-19, 3-35, 4-7, 6-8, 6-9, 6-14, 6-15, 7-4, 7-5, 7-11, 7-13, 9-15, 9-17, 9-18, 9-23, 9-39, 11-6, 13-10, 13-15, 13-25

input-output

1-2, 2-5, 6-13, 9-24, 9-25, 9-50, 13-26, 14-1 to 14-3, 15-2, 15-6 to 15-8, 16-13, 16-17

interfacing FORTRAN

2-4

internal codes

13-6

interrupt

5-2, 9-10, 9-30, 9-32, 9-33, 9-35, 9-36, 9-50, 9-54, 9-55, 13-14 to 13-19, 16-2

interrupt entries

13-14, 13-16

interrupt handler

9-30, 13-14, 13-16, 13-17, 13-19

interrupt priority

13-18

interrupt service routine

6-12

isam package

16-17

keyboard

6-2, 13-18, 14-2

library/libraries

1-2, 1-6, 2-4, 2-5, 3-3, 3-7, 3-8, 3-38, 3-43, 4-4, 4-11, 8-6, 9-6, 9-21, 9-50, 10-2 to 10-4, 10-8, 10-9, 10-11, 10-13, 10-15, 10-18, 11-4, 11-14, 12-3, 12-4, 13-3, 13-6, 13-12, 13-23, 13-24, 13-28, 15-2, 15-7, 15-8, 16-8

limited private types

5-6, 9-16, 11-5

limited types

3-22, 6-20, 7-10 to 7-13

lock files

14-5

KEY TECHNICAL TERMS

machine language

13-22

machine independent

4-4, 4-11

mantissa

4-4, 13-29

memory

1-5, 3-39 to 3-42, 4-17, 4-18, 16-2, 6-14, 16-18, 9-10, 9-12, 9-13, 9-16, 9-52, 9-54, 11-8, 11-9, 11-11, 11-12, 13-3 to 13-5, 13-12, 13-13, 13-23 to 13-25, 16-2, 16-18

memory access

9-54, 13-4, 13-5

memory allocation

13-12, 13-13, 15-10

memory reclamation

13-23

modularization

7-3

multi-tasking

9-28, 11-10

multiprocessor

5-2, 9-10, 9-15, 9-46, 16-9

mutation

3-47, 9-19

named associations

4-7, 6-16, 6-17

non-portable

1-2, 2-2, 3-39, 9-12, 9-20, 9-35, 9-44, 9-45, 11-4, 11-10

nonlimited

3-14, 3-19

nonlocal objects

6-7

null records

4-2

object oriented

3-5, 7-3, 7-10, 9-56

order dependence

1-4 to 1-6

order of elaboration

1-6, 3-8, 10-10, 10-18

output

1-2, 1-4, 2-5, 6-13, 9-24, 9-25, 9-50, 9-54, 10-17, 12-2, 13-9, 13-26, 14-1, 14-8, 15-2, 15-6 to 15-8, 16-13, 16-17

overflow

3-28 to 3-30, 4-26, 11-5, 11-6, 16-2, 16-16

overload

4-3, 6-22, 7-11, 9-10, 10-15

KEY TECHNICAL TERMS

package specification

7-2, 7-6, 7-7

parallel

6-3, 9-7, 15-2

parameter

1-5, 1-6, 3-10, 3-15, 3-37, 3-43, 3-45, 4-2, 4-23, 5-6, 5-7, 6-2, 6-4, 6-5, 6-7 to 6-10, 6-14, 6-16 to 6-18, 7-11, 9-2, 9-16, 11-4, 12-2, 13-7, 15-2, 15-6, 15-11, 16-5, 16-8

physical address

16-2

portable/portability

1-2, 2-2, 2-4, 2-5, 2-6, 3-2, 3-3, 3-34, 3-39, 4-9, 4-16, 5-3, 8-8, 8-9, 11-11, 16-11, 16-16, 16-18

pragma

3-2, 3-3, 3-40, 3-44, 6-3, 6-4, 6-12 to 6-14, 6-17, 9-4, 9-6, 9-12, 9-20, 9-21, 9-22, 9-25, 9-26, 9-28, 9-30, 9-35, 9-39, 9-44 to 9-46, 9-49, 9-52 to 9-55, 10-9, 10-12, 10-13, 10-18, 10-19, 11-4, 11-10, 13-4, 13-8 to 13-10, 13-20, 13-22, 13-23, 13-27, 15-9, 15-11, 16-11

priority

4-15 to 4-18, 9-2, 9-5, 9-6, 9-22, 9-25 to 9-27, 9-37, 9-39, 9-41 to 9-50, 13-3, 13-18, 13-19, 15-8, 16-15

private type

3-11, 3-14, 3-30, 4-2, 5-6, 7-2, 7-7, 7-9, 9-16, 11-6, 11-7

program states

9-52

program units

9-16, 16-5, 16-7, 16-8

put procedures

14-6, 14-8

qualified expression

4-14, 4-22

queues

9-2, 9-5, 9-23, 9-26, 9-37

race controls

16-15

raised exception

11-3, 11-4

readable/readability

2-3, 3-24, 4-14, 6-16, 11-2, 11-15, 11-16, 13-5, 16-3

real-time

3-2 to 3-4, 3-20, 3-33, 4-9, 9-4, 9-6, 9-9 to 9-11, 9-13, 9-14, 9-22, 9-24, 9-25, 9-27, 9-28, 9-39, 9-43 to 9-46, 9-50, 10-17, 11-3, 11-8, 11-11, 13-12, 13-13, 13-18 to 13-20, 13-23, 15-8, 16-11, 16-12, 16-15

record type

3-3, 3-37, 3-45, 8-4, 9-17, 9-55, 13-27, 15-6

recovery

5-2, 9-9, 9-10, 9-22, 11-8, 11-9

relation

4-8, 4-14

KEY TECHNICAL TERMS

rename

3-24, 8-3, 11-6

rendezvous4-11, 9-5, 9-15, 9-17, 9-21, 9-28, 9-36, 9-42, 9-43,
9-51, 9-52, 11-11**reserve**

11-8, 13-13

restart

1-4, 3-2, 3-41, 3-42, 4-4, 4-18,

runtime1-4, 3-2, 4-15, 4-18, 6-13, 6-14, 9-9 to 9-12, 9-20,
9-23, 9-24, 9-30, 9-32, 9-33, 9-38, 9-39, 9-45, 9-
48, 9-49, 9-54, 11-8, 11-11, 13-12, 13-14, 13-16, 13-
17, 13-20, 13-23**scheduling**9-22 to 9-24, 9-27, 9-33, 9-42, 9-43, 9-46, 13-17 to
13-19, 16-12, 16-15**scope**3-2, 3-7, 3-9, 3-43, 4-16, 5-8, 9-15, 9-22, 9-24, 11-
3, 11-4, 13-13, 13-14**selective wait**

9-35, 9-36

semicolon

10-14

sequential_io

13-27, 14-5, 15-2, 15-7

shared variables

9-54

shared generics

11-13

shift and rotate operation

4-9, 4-10

software repositories

1-2

software first

9-28, 13-10

standard package

11-4, 16-16

static expressions

3-2, 3-42, 4-22, 7-8

storage management

13-23

storage_error

11-8, 11-9, 11-11, 13-24

string

1-2, 3-40, 3-42, 4-2, 6-3, 6-22,

subprogram specification

6-7

subprogram address

6-14

subprogram call

6-10, 9-39, 11-9

KEY TECHNICAL TERMS

subunit

4-19, 4-20, 4-22, 4-23, 4-25
10-5, 10-14 to 10-16

suspended

9-39, 9-42

synchronization

7-6, 9-6, 9-17, 9-34, 9-36, 9-54, 9-55, 13-21

synchronous

15-11

system.Tick

9-30, 9-31

tasking

9-4-6, 9-12, 9-14, 9-18, 9-20, 9-22 to 9-25, 9-28, 9-29, 9-44, 9-50, 9-5213-18, 13-23, 11-7, 11-10, 15-11, 16-15

terminate

3-8, 9-21, 9-35, 9-38, 11-8, 13-24, 13-23

text_io

1-3, 1-4, 3-31, 4-20, 10-6, 10-7, 10-13, 10-18, 14-3, 14-4, 14-6, 15-7

tick

9-30, 9-31

timed entry calls

9-13

type conversion

3-28, 3-29, 4-19, 4-21, 4-22, 4-26, 8-6

type declarations

3-11, 7-7, 8-4, 8-6, 9-16, 16-7

unchecked conversion

3-29, 6-3, 9-52, 13-26 to 13-28

unchecked deallocation

4-16, 13-23, 13-24

unconstrained type

16-6

underscore

2-3

uninitialized

1-4

unsigned integer

3-29

unsigned number

16-2

use

1-6, 2-2, 2-4 to 2-6, 3-2, 3-3, 3-10, 3-13, 3-14, 3-16, 3-20, 3-23 to 3-25, 3-27 to 3-31, 3-33, 3-38 to 3-42, 3-45 to 3-476-2, 4-6, 4-7, 4-14, 4-16, 4-18 to 4-20, 4-22, 4-24, 5-2, 5-3, 5-8,
6-3, 6-12 to 6-14, 6-16, 6-17, 6-20, 6-23, 7-4 to 7-6, 7-8, 7-12 to 7-14, 8-2 to 8-4, 8-6 to 8-8, 9-2, 9-4, 9-5, 9-7, 9-10, 9-12 to 9-14, 9-17 to 9-19, 9-21 to 9-26, 9-28, 9-36, 9-42, 9-50 to 9-54, 10-4 to 10-6, 10-8, 10-9, 10-12 to 10-14, 10-16, 11-4, 11-8, 11-9, 11-11, 11-15, 12-2, 12-4, 13-2, 13-4, 13-5, 13-10, 13-13, 13-14, 13-22 to 13-24, 13-26, 13-28, 14-3, 15-3 to 15-7, 15-9, 15-10, 16-2 to 16-5, 16-9 to 16-13, 16-16, 16-18

variable

1-5, 2-7, 3-4, 3-6 to 3-8, 3-14, 3-19, 3-22, 3-23, 4-2, 4-18, 5-4, 5-6, 6-8, 6-22, 8-10, 9-7, 9-15, 9-55, 13-26, 14-4, 14-6, 16-17

KEY TECHNICAL TERMS

wake-up

9-33

with clause

3-43, 10-12

REVISION REQUEST BY NUMBER

<u>NUMBER</u>	<u>TITLE</u>	<u>PAGE</u>
0051	STANDARDIZATION OF GENERAL PURPOSE PACKAGES	1-2
0042	INCORRECT ORDER DEPENDENCIES	1-3
0066	ERRONEOUS EXECUTION AND INCORRECT ORDER DEPENDENCE	1-5
0034	USE 8-BIT ASCII	2-2
0126	UNDERSCORE BEFORE EXPONENT IN NUMERIC LITERALS	2-3
0039	INTERFACING FORTRAN LIBRARIES TO ADA PROGRAMS	2-4
0117	PRE-ELABORATION	3-2
0125	INTRODUCE INHERITANCE INTO ADA	3-5
0092	FINALIZATION	3-7
0094	IDENTIFIER LISTS AND THE EQUIVALENCE OF SINGLE AND MULTIPLE DECLARATIONS	3-10
0100	CONSTANTS CANNOT USE DEFAULT VALUES	3-14
0036	MAKE "EXCEPTION" A PREDEFINED TYPE	3-16
0129	INITIALIZATION FOR NONLIMITED TYPES	3-19
0058	NON-CONTIGUOUS SUBTYPES OF ENUMERATION TYPES	3-20
0019	VARIABLE FINALIZATION	3-22
0022	VISIBILITY OF BASIC OPERATIONS ON A TYPE	3-24
0052	MULTIPLE TYPE DERIVATIONS	3-25
0080	DERIVED TYPES	3-27
0045	OVERFLOW AND TYPE CONVERSION	3-28
0138	UNSIGNED INTEGERS	3-29
0122	LIMITATION ON RANGE OF INTEGER TYPES	3-30
0135	CATENATION OPERATION FOR ONE-DIMENSIONAL CONSTRAINED ARRAYS	3-31
0144	FLOATING POINT CO-PROCESSORS	3-33

REVISION REQUEST BY NUMBER

<u>NUMBER</u>	<u>TITLE</u>	<u>PAGE</u>
0061	FLOATING POINT MUST INCLUDE LONG_FLOAT AND SHORT_FLOAT	3-34
0086	REFERENCE TO SELF IN INITIAL VALUE EXPRESSION	3-35
0027	ADDITION OF ATTRIBUTES FOR RECORD TYPES	3-37
0018	STATIC RAGGED ARRAYS	3-38
0098	MUTUALLY DEPENDENT TYPES OTHER THAN ACCESS	3-45
0053	AGGREGATE FOR NULL RECORDS AND NULL ARRAYS	4-2
0024	SEPARATION OF EXPONENT AND MANTISSA	4-4
0059	ATTRIBUTE P'REPRESENTATION	4-6
0029	ALLOW OTHERS WITH NAMED ASSOCIATION AT ARRAY INITIALIZATION	4-7
0031	ALLOW RELATION TO SPECIFY NONCONTINUOUS RANGE	4-8
0139	SHIFT AND ROTATE OPERATIONS FOR BOOLEAN ARRAYS	4-9
0102	REMAINDER DIVIDE FOR REAL NUMBERS	4-11
0011	EXPONENTS OF ZERO BY A ZERO EXPONENT	4-13
0131	VISIBILITY OF HIDDEN IDENTIFIERS IN QUALIFIED EXPRESSIONS	4-14
0112	GARBAGE COLLECTION	4-15
0099	TYPE CONVERSION OF STATIC TYPE CAN BE NON-STATIC	4-19
0009	SOME CONVERSIONS SHOULD BE STATIC	4-22
0111	FAULT TOLERANCE	5-2
0049	REFERENCE TO VARIABLE NAMES	5-4
0088	USER DEFINED ASSIGNMENT STATEMENT FOR LIMITED PRIVATE TYPES	5-6
0104	ACCESSING A TASK OUTSIDE ITS MASTER	5-8
0064	SUBPROGRAM CALLBACK	6-2

REVISION REQUEST BY NUMBER

<u>NUMBER</u>	<u>TITLE</u>	<u>PAGE</u>
0128	SUBPROGRAMS AS PARAMETERS	6-4
0030	SUBPROGRAM SPECIFICATION	6-7
0002	READING OF OUT PARAMETERS	6-8
0055	SUBPROGRAM BODIES AS GENERIC INSTANTIATIONS	6-10
0060	PRAGMA SELECTIVE INLINE	6-12
0014	EXECUTION OF A PROGRAM UNIT BY ITS ADDRESS	6-13
0097	EXPLICIT INVOCATION OF DEFAULT PARAMETER	6-16
0026	MODE OF PARAMETERS OF A FUNCTION	6-18
0008	ALLOW OVERLOADING OF "="	6-20
0025	OVERLOADING "="	6-22
0082	COMPILATION UNITS	7-2
0140	PROBLEMS WITH OBJECT ORIENTED SIMULATION	7-3
0032	MUTUAL VISIBILITY REGION	7-4
0090	CONTROL OVER VISIBILITY OF TASK ENTRIES	7-5
0093	CONSTANTS DEFERRED TO PACKAGE BODY	7-7
0010	PRIVATE TYPE DERIVED FROM DISCRIMINATED TYPE	7-9
0070	USER DEFINED ASSIGNMENT	7-10
0001	LIMITED TYPES TOO LIMITED	7-12
0057	VISIBILITY OF OPERATORS BETWEEN PACKAGES	8-2
0096	LIMITATIONS ON USE OF RENAMING	8-4
0069	VISIBILITY CONTROL	8-6
0050	EXTENDED CHARACTER SET	8-8
0056	TASK PRIORITIES AND ENTRY FAMILIES	9-2
0084	REDUCING RUN-TIME TASKING OVERHEAD	9-4

REVISION REQUEST BY NUMBER

<u>NUMBER</u>	<u>TITLE</u>	<u>PAGE</u>
0133	ATTRIBUTES FOR TASK ARRAY COMPONENTS	9-7
0106	ASYNCHRONOUS EVENT HANDLING	9-9
0109	DISTRIBUTED SYSTEMS	9-12
0123	DISCRIMINATE VALUES PASSED AT TASK OBJECT CREATION	9-15
0012	MUTATION OF TYPES	9-19
0078	TASKING SEMANTICS	9-20
0023	TERMINATION OF TASKS	9-21
0016	ALTERNATE ADA TASK SCHEDULING	9-22
0015	USE OF TASK PRIORITIES IN ACCEPT AND SELECT STATEMENTS	9-25
0037	CONTROL OF CLOCK SPEED AND TASK DISPATCH RATE	9-28
0107	CONFIGURING CALENDAR.CLOCK IMPLEMENTATION	9-30
0108	DELAY UNTIL	9-32
0083	ASYNCHRONOUS TRANSFER OF CONTROL	9-35
0076	PRIORITY SELECT	9-37
0079	TERMINATE NOT USED	9-38
0013	DYNAMIC PRIORITIES FOR TASKS	9-39
0020	MODIFICATION OF TASK PRIORITIES DURING EXECUTION	9-41
0021	TASK SCHEDULING	9-42
0072	TASK PRIORITIES (ADA-UK/012)	9-43
0124	REVOKE AI-00594/02	9-45
0075	PRIORITY ENTRY QUEUING	9-47
0116	ALLOW MODIFIABLE PRIORITIES FOR TASKS	9-48
0151	PRAGMAS FOR TASK INTERRUPTS AND TIMED I/O	9-50
0134	COUNT ATTRIBUTE	9-51

REVISION REQUEST BY NUMBER

<u>NUMBER</u>	<u>TITLE</u>	<u>PAGE</u>
0063	ABORT STATEMENT	9-52
0119	SHARED COMPOSITE OBJECTS	9-54
0091	SECTION 10 SHOULD NOT DESCRIBE THE PROCESS OF COMPILATION, BUT RATHER THE MEANING OF PROGRAMS	10-2
0038	SUBUNIT NAMES	10-5
0073	GLOBAL NAME-SPACE CONTROL THROUGH MULTI-LEVEL PROGRAM LIBRARIES (ADA-UK/010)	10-8
0003	CLEANUP AFTER MAIN SUBPROGRAM	10-10
0095	CONTEXT CLAUSES AND APPLY	10-12
0028	ALLOW SEMICOLON AFTER SEPARATE CLAUSE	10-14
0041	ALLOW SUBUNITS WITH SAME ANCESTOR LIBRARY	10-15
0142	REDUCING COMPILATION COSTS	10-16
0004	TRANSITIVE PRAGMA ELABORATE	10-18
0033	PASS EXCEPTIONS AS PARAMETERS OR ACCESS EXCEPTION'S 'IMAGE	11-2
0085	FINDING THE NAME OF THE CURRENTLY RAISED EXCEPTION	11-3
0101	IMPLEMENTATION OF EXCEPTIONS AS TYPES	11-5
0118	PROVIDE USER-SPECIFIED STORAGE RESERVE FOR RECOVERY FROM STORAGE_ERROR	11-8
0145	RETRIEVE CURRENT EXCEPTION NAME	11-10
0120	HANDLING OF UNSUCCESSFUL ATTEMPTS TO ALLOCATE MEMORY	11-11
0005	EXCEPTION DECLARATIONS NOT SHARABLE	11-13
0132	"WHEN" CLAUSE TO RAISE EXCEPTIONS	11-15
0141	INCLUDE "WHEN" IN RAISE STATEMENT SYNTAX	11-16
0035	ALLOW OVERLOADING OF GENERIC PARAMETER STRUCTURES	12-2

REVISION REQUEST BY NUMBER

<u>NUMBER</u>	<u>TITLE</u>	<u>PAGE</u>
0006	UNCONSTRAINED SUBTYPES AS GENERIC ACTUALS	12-3
0065	BUILDING OBJECT PROGRAMS	13-2
0062	PROVIDING EXPLICIT CONTROL OF SIZE OF MEMORY ACCESS, I.E., BYTES, WORDS, LONG_WORDS.	13-4
0040	ADD ATTRIBUTE TO ACCESS INTERNAL CODE OF ENUMERATION LITERAL	13-6
0007	DEFAULT REPRESENTATION FOR ENUMERATION TYPES	13-8
0137	BIT/STORAGE UNIT ADDRESSING CONVENTION	13-10
0110	PROVIDE EXPLICIT CONTROL OF MEMORY USAGE	13-12
0114	ADDRESS CLAUSES AND INTERRUPT ENTRIES	13-14
0115	MODELS FOR INTERRUPT HANDLING	13-16
0087	TASKING PRIORITY INVERSION BECAUSE OF HARDWARE INTERRUPT	13-18
0105	SETTING/ADJUSTING CALENDAR.CLOCK	13-20
0043	USE OF ASSEMBLY LANGUAGE	13-22
0113	GUARANTEE MEMORY RECLAMATION	13-23
0103	LIMITATIONS OF UNCHECKED CONVERSION	13-26
0149	ADD PREDEFINED KEYBOARD I/O PACKAGE	14-2
0089	INTERACTIVE TERMINAL INPUT-OUTPUT	14-3
0054	DO NOT ADD VARIABLE STRING TYPE	14-4
0146	FILE AND RECORD LOCKING	14-5
0047	"GET" AND "PUT" AS FUNCTIONS	14-6
0130	DEFINE "DEFAULT_XY" IN IO PACKAGES AS FUNCTIONS	14-7
0127	OUTPUT OF REAL NUMBERS WITH BASES	14-8
0017	GENERIC INPUT-OUTPUT	15-2

REVISION REQUEST BY NUMBER

<u>NUMBER</u>	<u>TITLE</u>	<u>PAGE</u>
0068	SECONDARY STANDARDS FOR PREDEFINED LIBRARY UNITS (ADA-UK/006)	15-7
0148	USE FULL EXTENDED ASCII CHARACTER SET	15-9
0143	REQUIRED VENDOR DOCUMENTATION	15-10
0044	LEAVE UNSIGNED NUMBERS OUT	16-2
0046	EXTEND USE OF "&", " " AND KEYWORD "IS" AND "NOT IS"	16-3
0067	DEFINITION AND USE OF TECHNICAL TERMS ADA-UK/004	16-4
0071	APPLICABILITY TO DISTRIBUTE SYSTEMS (ADA-UK/007)	16-9
0074	RUN-TIME ENVIRONMENT DEFINITION AND INTERFACE	16-11
0077	STREAM I/O	16-13
0081	PACKAGE/SUBPROGRAM TYPES	16-14
0121	INCONSISTENCY IN ADA SEMANTICS OF RACE CONTROLS	16-15
0136	SUGGESTED STANDARD PACKAGE FOR BIT-LEVEL OPERATIONS ON INTEGER DATA TYPES	16-16
0147	ADD PREDEFINED ISAM PACKAGE	16-17
0150	PROVIDE CHAINING CAPABILITY IN PREDEFINED PROCEDURE	16-18

REVISION REQUEST BY TITLE

<u>TITLE</u>	<u>PAGE</u>
ABORT STATEMENT	9-52
ACCESSING A TASK OUTSIDE ITS MASTER	5-8
ADD ATTRIBUTE TO ACCESS INTERNAL CODE OF ENUMERATION LITERAL	13-6
ADD PREDEFINED ISAM PACKAGE	16-17
ADD PREDEFINED KEYBOARD I/O PACKAGE	14-2
ADDITION OF ATTRIBUTES FOR RECORD TYPES	3-37
ADDRESS CLAUSES AND INTERRUPT ENTRIES	13-14
AGGREGATE FOR NULL RECORDS AND NULL ARRAYS	4-2
ALLOW MODIFIABLE PRIORITIES FOR TASKS	9-48
ALLOW OTHERS WITH NAMED ASSOCIATION AT ARRAY INITIALIZATION	4-7
ALLOW OVERLOADING OF "="	6-20
ALLOW OVERLOADING OF GENERIC PARAMETER STRUCTURES	12-2
ALLOW RELATION TO SPECIFY NONCONTINUOUS RANGE	4-8
ALLOW SEMICOLON AFTER SEPARATE CLAUSE	10-14
ALLOW SUBUNITS WITH SAME ANCESTOR LIBRARY	10-15
ALTERNATE ADA TASK SCHEDULING	9-22
APPLICABILITY TO DISTRIBUTE SYSTEMS (ADA-UK/007)	16-9
ASYNCHRONOUS EVENT HANDLING	9-9
ASYNCHRONOUS TRANSFER OF CONTROL	9-35
ATTRIBUTE P'REPRESENTATION	4-6
ATTRIBUTES FOR TASK ARRAY COMPONENTS	9-7
BIT/STORAGE UNIT ADDRESSING CONVENTION	13-10

REVISION REQUEST BY TITLE

<u>TITLE</u>	<u>PAGE</u>
BUILDING OBJECT PROGRAMS	13-2
CATENATION OPERATION FOR ONE-DIMENSIONAL CONSTRAINED ARRAYS	3-31
CLEANUP AFTER MAIN SUBPROGRAM	10-10
COMPILATION UNITS	7-2
CONFIGURING CALENDAR.CLOCK IMPLEMENTATION	9-30
CONSTANTS CANNOT USE DEFAULT VALUES	3-14
CONSTANTS DEFERRED TO PACKAGE BODY	7-7
CONTEXT CLAUSES AND APPLY	10-12
CONTROL OF CLOCK SPEED AND TASK DISPATCH RATE	9-28
CONTROL OVER VISIBILITY OF TASK ENTRIES	7-5
COUNT ATTRIBUTE	9-51
DEFAULT REPRESENTATION FOR ENUMERATION TYPES	13-8
DEFINE "DEFAULT_XY" IN IO PACKAGES AS FUNCTIONS	14-7
DEFINITION AND USE OF TECHNICAL TERMS ADA-UK/004	16-4
DELAY UNTIL	9-32
DERIVED TYPES	3-27
DISCRIMINATE VALUES PASSED AT TASK OBJECT CREATION	9-15
DISTRIBUTED SYSTEMS	9-12
DO NOT ADD VARIABLE STRING TYPE	14-4
DYNAMIC PRIORITIES FOR TASKS	9-39
ERRONEOUS EXECUTION AND INCORRECT ORDER DEPENDENCE	1-5
EXCEPTION DECLARATIONS NOT SHARABLE	11-13

REVISION REQUEST BY TITLE

<u>TITLE</u>	<u>PAGE</u>
EXECUTION OF A PROGRAM UNIT BY ITS ADDRESS	6-13
EXPLICIT INVOCATION OF DEFAULT PARAMETER	6-16
EXPONENTS OF ZERO BY A ZERO EXPONENT	4-13
EXTEND USE OF "&", " " AND KEYWORD "IS" AND "NOT IS"	16-3
EXTENDED CHARACTER SET	8-8
FAULT TOLERANCE	5-2
FILE AND RECORD LOCKING	14-5
FINALIZATION	3-7
FINDING THE NAME OF THE CURRENTLY RAISED EXCEPTION	11-3
FLOATING POINT CO-PROCESSORS	3-33
FLOATING POINT MUST INCLUDE LONG_FLOAT AND SHORT_FLOAT	3-34
GARBAGE COLLECTION	4-15
GENERIC INPUT-OUTPUT	15-2
"GET" AND "PUT" AS FUNCTIONS	14-6
GLOBAL NAME-SPACE CONTROL THROUGH MULTI-LEVEL PROGRAM LIBRARIES (ADA-UK/010)	10-8
GUARANTEE MEMORY RECLAMATION	13-23
HANDLING OF UNSUCCESSFUL ATTEMPTS TO ALLOCATE MEMORY	11-11
IDENTIFIER LISTS AND THE EQUIVALENCE OF SINGLE AND MULTIPLE DECLARATIONS	3-10
IMPLEMENTATION OF EXCEPTIONS AS TYPES	11-5
INCLUDE "WHEN" IN RAISE STATEMENT SYNTAX	11-16
INCONSISTENCY IN ADA SEMANTICS OF RACE CONTROLS	16-15
INCORRECT ORDER DEPENDENCIES	1-3

REVISION REQUEST BY TITLE

<u>TITLE</u>	<u>PAGE</u>
INITIALIZATION FOR NONLIMITED TYPES	3-19
INTERACTIVE TERMINAL INPUT-OUTPUT	14-3
INTERFACING FORTRAN LIBRARIES TO ADA PROGRAMS	2-4
INTRODUCE INHERITANCE INTO ADA	3-5
LEAVE UNSIGNED NUMBERS OUT	16-2
LIMITATION ON RANGE OF INTEGER TYPES	3-30
LIMITATIONS OF UNCHECKED CONVERSION	13-26
LIMITATIONS ON USE OF RENAMING	8-4
LIMITED TYPES TOO LIMITED	7-12
MAKE "EXCEPTION" A PREDEFINED TYPE	3-16
MODE OF PARAMETERS OF A FUNCTION	6-18
MODELS FOR INTERRUPT HANDLING	13-16
MODIFICATION OF TASK PRIORITIES DURING EXECUTION	9-41
MULTIPLE TYPE DERIVATIONS	3-25
MUTATION OF TYPES	9-19
MUTUAL VISIBILITY REGION	7-4
MUTUALLY DEPENDENT TYPES OTHER THAN ACCESS	3-45
NON-CONTIGUOUS SUBTYPES OF ENUMERATION TYPES	3-20
OUTPUT OF REAL NUMBERS WITH BASES	14-8
OVERFLOW AND TYPE CONVERSION	3-28
OVERLOADING "="	6-22
PACKAGE/SUBPROGRAM TYPES	16-14
PASS EXCEPTIONS AS PARAMETERS OR ACCESS EXCEPTION'S 'IMAGE	11-2

REVISION REQUEST BY TITLE

<u>TITLE</u>	<u>PAGE</u>
PRAGMA SELECTIVE INLINE	6-12
PRAGMAS FOR TASK INTERRUPTS AND TIMED I/O	9-50
PRE-ELABORATION	3-2
PRIORITY ENTRY QUEUING	9-47
PRIORITY SELECT	9-37
PRIVATE TYPE DERIVED FROM DISCRIMINATED TYPE	7-9
PROBLEMS WITH OBJECT ORIENTED SIMULATION	7-3
PROGRAMS	10-2
PROVIDE CHAINING CAPABILITY IN PREDEFINED PROCEDURE	16-18
PROVIDE EXPLICIT CONTROL OF MEMORY USAGE	13-12
PROVIDE USER-SPECIFIED STORAGE RESERVE FOR RECOVERY FROM STORAGE_ERROR	11-8
PROVIDING EXPLICIT CONTROL OF SIZE OF MEMORY ACCESS, I.E., BYTES, WORDS, LONG_WORDS.	13-4
READING OF OUT PARAMETERS	6-8
REDUCING COMPILATION COSTS	10-16
REDUCING RUN-TIME TASKING OVERHEAD	9-4
REFERENCE TO SELF IN INITIAL VALUE EXPRESSION	3-35
REFERENCE TO VARIABLE NAMES	5-4
REMAINDER DIVIDE FOR REAL NUMBERS	4-11
REQUIRED VENDOR DOCUMENTATION	15-10
RETRIEVE CURRENT EXCEPTION NAME	11-10
REVOKE AI-00594/02	9-45
RUN-TIME ENVIRONMENT DEFINITION AND INTERFACE	16-11

REVISION REQUEST BY TITLE

<u>TITLE</u>	<u>PAGE</u>
SECONDARY STANDARDS FOR PREDEFINED LIBRARY UNITS (ADA-UK/006)	15-7
SECTION 10 SHOULD NOT DESCRIBE THE PROCESS OF COMPILATION, BUT RATHER THE MEANING OF SEPARATION OF EXPONENT AND MANTISSA	4-4
SETTING/ADJUSTING CALENDAR.CLOCK	13-20
SHARED COMPOSITE OBJECTS	9-54
SHIFT AND ROTATE OPERATIONS FOR BOOLEAN ARRAYS	4-9
SOME CONVERSIONS SHOULD BE STATIC	4-22
STANDARDIZATION OF GENERAL PURPOSE PACKAGES	1-2
STATIC RAGGED ARRAYS	3-38
STREAM I/O	16-13
SUBPROGRAM BODIES AS GENERIC INSTANTIATIONS	6-10
SUBPROGRAM CALLBACK	6-2
SUBPROGRAM SPECIFICATION	6-7
SUBPROGRAMS AS PARAMETERS	6-4
SUBUNIT NAMES	10-5
SUGGESTED STANDARD PACKAGE FOR BIT-LEVEL OPERATIONS ON INTEGER DATA TYPES	16-16
TASK PRIORITIES (ADA-UK/012)	9-43
TASK PRIORITIES AND ENTRY FAMILIES	9-2
TASK SCHEDULING	9-42
TASKING PRIORITY INVERSION BECAUSE OF HARDWARE INTERRUPT	13-18
TASKING SEMANTICS	9-20
TERMINATE NOT USED	9-38
TERMINATION OF TASKS	9-21

REVISION REQUEST BY TITLE

<u>TITLE</u>	<u>PAGE</u>
TRANSITIVE PRAGMA ELABORATE	10-18
TYPE CONVERSION OF STATIC TYPE CAN BE NON-STATIC	4-19
UNCONSTRAINED SUBTYPES AS GENERIC ACTUALS	12-3
UNDERSCORE BEFORE EXPONENT IN NUMERIC LITERALS	2-3
UNSIGNED INTEGERS	3-29
USE 8-BIT ASCII	2-2
USE FULL EXTENDED ASCII CHARACTER SET	15-9
USE OF ASSEMBLY LANGUAGE	13-22
USE OF TASK PRIORITIES IN ACCEPT AND SELECT STATEMENTS	9-25
USER DEFINED ASSIGNMENT	7-10
USER DEFINED ASSIGNMENT STATEMENT FOR LIMITED PRIVATE TYPES	5-6
VARIABLE FINALIZATION	3-22
VISIBILITY CONTROL	8-6
VISIBILITY OF BASIC OPERATIONS ON A TYPE	3-24
VISIBILITY OF HIDDEN IDENTIFIERS IN QUALIFIED EXPRESSIONS	4-14
VISIBILITY OF OPERATORS BETWEEN PACKAGES	8-2
"WHEN" CLAUSE TO RAISE EXCEPTIONS	11-15

REVISION REQUEST BY SUBMITTER

ACM Special Interest Group on Ada

3-2, 4-15, 5-2, 5-8, 9-9, 9-12, 9-30, 9-32, 9-45, 9-48, 9-54, 11-8, 11-11, 13-12, 13-14, 13-16, 13-20, 13-23

Ada Europe Environment Working Group

9-9, 9-12, 9-30, 9-32, 9-45, 9-48, 9-54, 10-22, 14-3

Ada UK

1-5, 3-7, 6-2, 7-10, 8-6, 9-43, 10-2, 10-8, 13-2, 15-7, 16-4, 16-9, 16-11

Allison, James F.

3-22

Altman, Neal

4-15

Baker, Ted

3-2, 3-35, 9-4, 9-9, 9-54, 13-22, 13-26

Barnes, J. G. P.

7-10, 9-43

Battany, David M.

4-13

Ben-Ari, M.

3-33, 10-16, 15-10

Brookman, David

3-24, 6-10, 9-21, 9-41, 9-42

Bryant, Richard

6-13

Carter, Jeffrey R.

4-14, 9-7, 11-15

Clarke, Chris

14-2, 14-5, 15-9, 16-17, 16-18

Clarson, Donald R.

3-30, 3-31, 15

Curtis, Mike

14-3

Dawes, John

8-6, 15-7, 16-4

Elrad, Tzilla

16-15

Engelson, Arny B.

10-15, 13-6

Eric F. Heck

6, 6-12

Farrington, K. M.

7-5

Fasano, Joseph

3-34

French, Stewart

3-17, 9-20, 9-38, 16-14

Gallaher, Lawrence J.

4-4, 4-11

REVISION REQUEST BY SUBMITTER

Garlington, Ken
3-38

Lehman, Larry
5-2

Gart, Mitch
11-10

Lester, C.
3-7

Goldenberg, Joanne
52

Levine, Gertrude
9-51, 9-52

Goranson, H.T.
9-19

Lewis, Dennis
9-50

Grein, Christoph
5-6

Lieberman, Deb
7-2

Heck, Eric F.
3-20

Lyons, T.G.L.
10-2

Hedstrom, John
16-13

MacLaren, Lee
15-2

High Integrity Systems
8-2

Orme, Tony
11-3

Kamrad, Mike
13-16

Page, Robert
13-18

Kellogg, Cliver M.
16-16

Papay, David
11-5

Komatar, Mark F.
1-2

Pazy, Offer
11-11, 13-12, 13-23

Lease, Damon
3-16

Pickett, M.J.
10-8, 16-11

REVISION REQUEST BY SUBMITTER

Pogge, R. David
10-5

Power, Kent
9-22, 9-25

Powers, Richard D.
9-30, 9-32, 9-37, 9-47
Quiggle, Tom
13-14

Racine, Roger
9-45

Roark, Chuck
9-39

Rose, Kjell
7-3

Salant, Neil
3-29, 4-9, 13-10

Sercely, Ronald
13-4

Stock, Dan
5-8, 9-4, 11-8

Taft, S. Tucker
4-22, 6-8, 6-20, 7-9, 7-12, 9-35, 10-10, 10-18, 11-13, 12-3, 13-8

Taylor, Bill
6-2, 13-2

Thomas, E.N.
3-10, 3-14, 3-45, 4-19, 6-16, 7-7, 8-4, 10-12

Tooby, B. C.
8-2, 9-2

Van der Laan, C. G.
2-4

Victor, Michael
9-48

Weber, Mats
1-3, 3-25, 4-2

Wellings, A.J.
9-12

Wichmann, B.A.
1-5

Williams, Gilbert T.
9-28

Winkler, Jurgen F H
2-3, 3-5, 3-19, 6-4, 14-7, 14-8

Wolfe, William F.
2-2, 3-37, 4-7, 4-8, 5-4, 6-7, 6-18, 6-22, 7-4, 10-14, 11-2, 11-16, 12-2, 14-6,

Wong, Sy
3-28, 8-8, 13-22, 14-4, 16-2, 16-3

REVISION REQUEST BY ORGANIZATION

Aerospace GCSD
4-6

Ferranti Computer Systems Limited
6-2, 13-2

AFATL/FXG
9-50

Florida State University
3-2, 3-35, 9-4, 9-9, 9-54, 13-20, 13-26

Alsys Ltd
7-10, 9-43, 11-10, 16-9

GE Aerospace GCSD
3-20, 6-12

AT&T Bell Laboratories
10-15, 13-6

General Dynamics-Data Systems Division
3-38

ATM Computer
11-3

Giordano Associates, Inc.
9-52

Berkshire, Fox & Associates
14-2, 14-5, 15-9, 16-17, 16-18

GTE Government Systems Corp
3-16, 11-5

Boeing
9-22, 9-25, 15-2

Hauptstr. 42
5-6

Brandeis University
3-33, 10-16, 15-10

High Integrity Systems
8-2, 9-2

C.S. Draper Laboratory
9-45

ICL
8-6, 14-3, 15-7, 16-4

Clemson University
2-2, 3-37, 4-7, 4-8, 5-4, 6-18, 6-22, 6-7, 7-4, 10-14, 11-16, 11-2, 12-2, 14-6

Illinois Institute of Technology
16-15

Delco Systems Operation
6-13

Integrated Systems Inc.
5-2

Fairleigh Dickinson University
9-51, 9-52

Intermetrics, Inc.
4-22, 6-8, 6-20, 7-9, 7-12, 9-35, 10-10, 10-18, 11-13, 12-3, 13-8

REVISION REQUEST BY ORGANIZATION

ITT Avionics
3-29, 4-9, 13-10

Rekenentrum der Rijksuniversiteit
2-4

LMSC 5T 40
4-4, 4-11

Science Applications International Corp.
9-19

Lockheed Missiles and Space Company
3-22

SD-Scicon PLC
3-10, 3-14, 3-45, 4-19, 6-16, 7-7, 8-4, 10-12

Magnavox Electronic Systems Company
3-24, 6-10, 9-21, 9-41, 9-42

SEMA Group (UK)
7-5, 10-8, 16-11

Martin Marietta Astronautics Group
4-14, 9-7, 11-15

Siemens AG ZFE F2 SOF3
2-3, 3-5, 3-19, 6-4, 14-7, 14-8

MBB BmbH
16-16

Software Engineering Institute
4-15

National Physical Laboratory
1-5

Software Leverage Inc.
13-12, 13-23

Naval Weapons Center
10-5, 13-18

Software Sciences Limited
10-2

NDRE
7-3

Swiss Federal Institute of Technology
1-3, 3-25, 4-2

Portsmouth Polytechnic
3-7

TASC
13-4

R.R. Software, Inc.
5-8, 9-4, 11-8

Teledyne Brown Engineering
3-30, 3-31, 9-15

Raytheon Company
9-48

TeleSoft
13-14

REVISION REQUEST BY ORGANIZATION

Texas Instruments

3-27, 7-2, 9-20 9-30, 9-32, 9-37 to 9-39, 9-47, 16-13, 16-14

U.S. Army Management Engineering College

1-2

Unisys Computer Systems Division

13-16

University of York

9-12

Westinghouse Electric Corporation

9-28